session 17 - multi-threading

We are going to look at multi-threading in c++. Historically, multi-threading capabilities were provided by
os-specific libraries. However, with c++11, we now have the `threading` library. We are going to take a look at
how to use this library presently. But first, lets talk about threading in general.

Concurrency is the ability of a process to make progress on more than one task; this could happen in parallel, as is
the case when running multiple threads on multiple cores. Or, this could happen by switching repeatedly between
tasks, providing forward progress on all of them, regardless of their complexity.

A couple of things to note. In python, we don't typically make use of threading, as Python has something called the
GIL, which should make a bit more sense after this session. Instead, Python provides a multi-processing model via
pyprocessing.

Threads vs Processes in Concurrent Programming

A process is an execution context with its own environment, memory, processor state, etc. Communication between
processes is handled via message passing, as is synchronization. A process is a heavy weight entity which takes a
long time to create in relation to a thread.

A thread is a light weight sequence of instructions which can be executed concurrently with other sequences in a
multi threading environment, while sharing the same address space. Because threads share the same address space, much
more care must be taken when programming them, lest they stomp on each other. However, due to their light weight
nature, threads are very cheep to create compared to processes. And because they share a unified address space, one
does not have to employ complicated messaging frameworks to communicate between them.

But enough babbling. Lets jump in and take a look at threads in action. Starting with some boilerplate…

```
#include <iostream>
#include <thread>

int main() {
```

```
    simple_thread();
    return 0;
}
```

As you can see, we need to include `thread`. This will become important when we implement the `simple_thread`
function. Lets do that now.

```
void simple_thread() {
    std::cout << 'I am a thread!" << std::endl;
}
```

And lets use this function in main.

First we are going to need to create a thread. We do this with the `std::thread` constructor, which can take a
function. Once we create a thread, its payload starts running. Next, we need to get the main thread to wait for t1 to
finish. We do this by calling `join` on the thread. You may only call join once on a thread. Otherwise, you will
crash your program. This isn't a problem in our trivial examples, but it can be in longer programs. In order to
help you out, threads provide the `joinable` method to test for this.

A thread may also be `detatched` instead of joined, if you have no need to synchronize execution. However, if
your main thread finishes before your detatched thread, your program will terminate.

```
int main() {
    // initialize the thread
    std::thread t1(simple_thread); // t1 starts running.

    // wait for it to finish
    if( t1.joinable() )
        t1.join();

}
```

Our example is incredibly simple. The main thread doesn't do anything but wait. Clearly, there is no reason to even
use threading with a problem this simple. Let's add some work in the main thread.

```
int main() {
    std::thread t1(function_simple);

    for(int i=0; i<100; i++) {
        std::cout << "from main " << i << std::endl;
    }

    t1.join();
    return 0;
}
```

Believe it or not, even in this trivial program, we already have a potential problem. What happens if an exception gets thrown from main before we call t1.join ? This could spell trouble as we wont be able to clean up. Let's fix this.

We are going to catch any thrown errors in our main thread, and, if we do catch an error, we are going to call join on t1, and then re-throw the caught error.

```
int main() {
    std::thread t1(function_simple);

    try {
        for(int i=0; i<100; i++) {
            std::cout << "from main " << i << std::endl;
        }
    } catch (...) {
        t1.join();
        throw;
    }

    t1.join();

    return 0;
}
```

Of course, we could avoid this by using RAII to create a class which wraps t1, and which calls join if necessary when the wrapper instance is destroyed. However, I am going to leave that for an exercise…

The thread constructor can take any callable as a parameter. That means we can pass it a functor. Lets define a simple functor and play a bit.

```
class Fctor {
public:
    void operator()() {
        for(int i =0; i>-100; i-- ) {
            std::cout << "from Fctor " << i << std::endl;
        }
    }
};
```

And lets use it

```
int main() {
    Functor fct;

    std::thread t1(fct);
     try {
        for(int i=0; i<100; i++) {
            std::cout << "from main " << i << std::endl;
        }
    } catch (...) {
        t1.join();
        throw;
    }

    t1.join();
    return 0;
}
```

When we run this, we get quite a mess. But before we look at cleaning this up, lets take a look at how to pass
parameters to the wrapped function.

```
class Fctor {
public:
    void operator()(string msg) {
        for(int i =0; i>-100; i-- ) {
            std::cout << "from Fctor says " << msg << std::endl;
        }
    }
};
```

Here is how we pass the string:

```
int main() {
```

```
    Functor fct;

    std::string s = "I am a thread!";
    std::thread t1(fct, s);
     try {
        for(int i=0; i<100; i++) {
            std::cout << "from main " << i << std::endl;
        }
    } catch (...) {
        t1.join();
        throw;
    }

    t1.join();
    return 0;
}
```

What if we want to pass that string by reference in order to be a bit more efficient? Well, threads pass data by
value. If we simply do the following, it won't behave as expected:

```
void operator()(std::string& msg){
    ...
}
```

If we really want to do this, we have to use a reference wrapper when passing the string.

```
std::thread t1(fct, std::ref(s) );
```

Another thing to be aware of. If we try and get a bit more terse and construct the functor in the thread constructor,
we will run into "c++'s most vexing parse". Basically , the following will not do what we want:

```
std::thread t1(Fctor(), s);
```

Instead, we are going to have to wrap the Functor instantiation in a pair of parens:

```
std::thread t1((Fctor()), s);
```

Threads cannot be copied. If you need to transfer ownership of a thread, you must `std::move` it.

thread ids

Threads all have unique ids. You can call `get_id()` on a thread to retrieve it's id. You can retrieve the id of the parent thread using `std::this_thread::get_id()`.

cpu availability

Under normal circumstances, you should avoid creating more threads than you have processors to handle them. In order to determine what that number is programmatically, you can call `std::thread::hardware_concurrency()`.

Example - putting it all together

```
class Fctor {
public:
    void operator()(std::string& ;msg) {
        for(int i =0; i>-100; i-- ) {
            std::cout << "from Fctor " << i << " " << msg << std::endl;
        }
    }
};

int main() {
    std::cout << "Number of available procs " << std::thread::hardware_concurrency() << std:
    std::cout << "Main thread's id " << std::this_thread::get_id() << std::endl;

    std::string s = "I am a thread!!!!";
    std::thread t1((Fctor()),std::move(s));

    std::cout << "t1 id " << t1.get_id() << std::endl;

    try {
        for(int i=0; i<100; i++) {
            std::cout << "From main " << i << std::endl;
        }
    } catch (...) {
        t1.join();
        throw;
    }

    t1.join();
```

```
    return 0;
}
```

Data Races and What to Do About Them

In our previous example, you undoubtedly noticed that the output between the t1 thread and the main thread was
interleaved. Why did this happen? Both threads were competing for a common resource - cout. We can fix this by using
a *mutex*.

Mutex stands for **mu**tual **ex**clusion object. A mutex is an os primitive which is used to protect a resource.

Let's return to our previous example and fix it up a bit.

```
#include <thread>
#include <iostream>
#include <mutex>

using namespace std;


void shared_print(string msg, int id) {

}

void function_1() {
    for(int i =0; i<100; i++) {
        shared_print(string("from t1:", i);
    }
}

int main() {
    std::thread t1(function_1);

    for(int i=0; i<100; i++) {
        shared_print(string("From main: "), i);
    }

    t1.join();

    return 0;
}
```

Ok Let's implement shared_print now. first, I create a global mutex. Then i define the function.

```
mutex mu;

void shared_print(string msg, int id) {
    mu.lock();
    cout << msg << " " << id << endl;
    mu.unlock();
}
```

Now `cout` in shared_print is protected. Lets run this.

Problem - exceptions

As before, we have an issue if the code between mu.lock and mu.unlock throws an exception. We could add a try catch
block or implement a wrapper via RAII, but the library does this for us already.

```
std::mutex mu;

void shared_print(string msg, int i) {
    lock_guard<mutex> guard(mu); // RAII
    cout << msg << " " << id << endl;
}
```

Problem - cout still callable

We generally want to package the mutex with the resource we are trying to protect in order to make it impossible to
access the resource directly. Here is an example class which handles this:

```
class LogFile {
    std::mutex m_mutex;
    ofstream f;
public:
    LogFile() {
        f.open("/tmp/log.txt");
    }

    void shared_print(string id, string msg) {
        std::lock_guard<std::mutex> locker(m_mutex);
        f << "From " << id << ": " << value << endl;
```

```
    }
};

void function_1(LogFile& log) {
    for(int i=0; i>-100; i--) {
        log.shared_print(i, string("From t1"));
    }
}

int main() {
    LogFile log;
    std::thread t1(function_1, std::ref(log));

    for(int i=0; i<100; i++) {
       log.shared_print(i, string("From Main"));
    }

    t1.join();
    return 0;
}
```

Note that you need to make certain that your class truly protects the resource under guard. For instance, if you
implement a method which returns a reference to f (the ofstream). And never allow a user defined function to operate
on f. Take care!

This is important.