Concurrency in C++11 series

pabloxrl@gmail.com

Introduction

- Purposes:
  - Bring everyone up to speed with C++11 Multithreading/Concurrency facilities
  - Provide insightful examples on how to use new C++11 features
- Different topics will be covered in this series incrementally:
  - Basic thread management
  - Thread-safe data structures
  - Lock-free data structures
  - And more

Introduction

- Baseline (or what do I assume you know):
  - What is concurrency
  - How multithreading differs from task switching
  - The difference between concurrency with multiple processes and concurrency with mutliple threads
  - Why (and why not) to use concurrency
- Sources:
  - Main source: C++ Concurrency in Action
  - Other sources will be quoted
- Code will be available in my github repository;
  - https://github.com/pabloribalta/Concurrency.git

C++11: Concurrency and multithreading

- The 1998 C++ standard doesn't acknowledge the existance of threads
  - Compilers had to ship extensions to enable multithreading
  - Prevalence of C APIs for fultithreading
    - POSIX C standard
    - Microsoft Windows API
  - General purpose C++ libaries like Boost and ACE tried to conceal platform-specifics
- Ultimately: No support in the standard

C++11: Concurrency and multithreading

- C++11 includes a new thread-aware memory model and the C++ Standard Library has been extended
  - Boost Thread Library as a primary model (names, structure, and such)
- This comes at a cost of *abstraction penalty b*ut low level facilities are also provided

Hello, concurrent world

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

void hello()
{
  std::cout << "Hello concurrent world\n";
}

int main()
{
  std::thread t(hello);
  t.join();

  return 0;
}
```

Hello, concurrent world (s1t01)


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

void hello()
{
  std::cout << "Hello concurrent world\n";
}

int main()
{
  std::thread t(hello);
  t.join();

  return 0;
}
```

Basic thread management

- Launching threads
  - Boils down to constructing an `std::thread` object

```
void do_some_work(){ ... }
std::thread my_thread(do_some_work);
```

  - `std::thread` works with every *callable* type, so it can accept an instance of a class with a function call operator instead (s1t02)
  - Lambda functions can also be used, which can help to avoid the common problem dubbed as the „C++ most vexing parse” (s1t03)

Basic thread management

- Waiting for threads to complete
  - Explicity choose what to do with the thread
    - Wait for it to complete (`myThread.join()`)
    - Let it run free (`myThread.detach()`)
  - Failing to do so before the `std::thread` object is destroyed will result in a crash (`std::thread` destructor calls `std::terminate`)
  - When not waiting for the thread completion, we must ensure that the data accessed by the thread is valid until the thread has finished with it. (s1t04)

Basic thread management

- Waiting in exceptional circumstances:
  - Avoiding termination when an exception is thrown using a try/catch block (s1t05)
  - Using RAII to wait for a thread to complete (s1t06)
  - If it is not necessary to wait for a thread to finish, we can *detach* from ensuring that `std::terminate()` won't be called when `std::thread` is destroyed.

Basic thread management

- Passing arguments to a thread
  - **Common pitfall:** By default, arguments are *copied* into internal storage, where they can be accessed by the newly created thread of execution, even if the parameter should be a reference. (s1t07 and s1t08)
  - Semantics are similar to `std::bind` when using member functions. (s1t09)
  - It is possible to use `std::move` when passing non-copyable objects as parameters (s1t10)

Basic thread management

- Transferring ownership of a thread
  - Keeping in mind that `std::thread` is *movable* but not *copyable* (s1t11)
  - Scenario #0: A function that creates a thread to run in the background and passes back the ownership to its caller (s1t12)
  - Scenario #1: Create a thread and pass its ownership in to some function that should wait for it (s1t12)
  - Possible to improve our ThreadGuard (s1t13)

Basic thread management

- High level thread management
  - Determining number of runnable parallel threads with `std::thread::hardware_concurrency`
  - Identifying threads with `std::thread_id` (s1t14)
  - **Bonus:** A naive parallel version of `std::accumulate` (s1t15)

Basic thread management

- Summary
  - Starting threads
  - Waiting (and *not waiting*) for threads to finish
  - Running threads in the background
  - Passing arguments to threads
  - Transfer of ownership of `std::thread` objects
  - How to manage a group of threads

Next presentation

- Sharing data between threads
  - Avoiding race conditions
  - Using std::mutex and std::lock
  - Structuring code for protecting shared data
  - Flexible locking strategies
  - Transfering ownership of locks and mutexes between scopes
  - Alternative facilities for protecting shared data

The end

- Any questions?