C++11 Concurrency Tutorial

Anthony Williams

Just Software Solutions Ltd
http://www.justsoftwaresolutions.co.uk

28th April 2012

- Asynchronous tasks and threads
- Promises and tasks
- Mutexes and condition variables
- Atomics

Spawning asynchronous tasks

Spawning asynchronous tasks

- Two ways: `std::async` and `std::thread`
- It's all about things that are **Callable**:
  - Functions and Member functions
  - Objects with `operator()` and Lambda functions

Hello World with std::async

```
#include <future> // for std::async
#include <iostream>
void write_message(std::string const& message) {
    std::cout<<message;
}
int main() {
    auto f=std::async(write_message,
        "hello world from std::async\n");
    write_message("hello world from main\n");
    f.wait();
}
```

Hello World with std::thread

```
#include <thread> // for std::thread
#include <iostream>
void write_message(std::string const& message) {
    std::cout<<message;
}
int main() {
    std::thread t(write_message,
        "hello world from std::thread\n");
    write_message("hello world from main\n");
    t.join();
}
```

Missing `join` with `std::thread`

```
#include <thread>
#include <iostream>
void write_message(std::string const& message) {
    std::cout<<message;
}
int main() {
    std::thread t(write_message,
        "hello world from std::thread\n");
    write_message("hello world from main\n");
    // oops no join
}
```

Missing `wait` with `std::async`

```
#include <future>
#include <iostream>
void write_message(std::string const& message) {
    std::cout<<message;
}
int main() {
    auto f=std::async(write_message,
        "hello world from std::async\n");
    write_message("hello world from main\n");
    // oops no wait
}
```

Async Launch Policies

- The standard launch policies are the members of the `std::launch` scoped enum.
- They can be used individually or together.

Async Launch Policies

- `std::launch::async` => "as if" in a new thread.
- `std::launch::deferred` => executed on demand.
- `std::launch::async | std::launch::deferred` => implementation chooses (default).

std::launch::async

```
#include <future>
#include <iostream>
#include <stdio.h>
void write_message(std::string const& message) {
  std::cout<<message;
}
int main() {
  auto f=std::async(
    std::launch::async,write_message,
    "hello world from std::async\n");
  write_message("hello world from main\n");
  getchar();  f.wait();
}
```

std::launch::deferred

```
#include <future>
#include <iostream>
#include <stdio.h>
void write_message(std::string const& message) {
  std::cout<<message;
}
int main() {
  auto f=std::async(
    std::launch::deferred,write_message,
    "hello world from std::async\n");
  write_message("hello world from main\n");
  getchar();  f.wait();
}
```

Returning values with `std::async`

```
#include <future>
#include <iostream>
int find_the_answer() {
    return 42;
}

int main() {
    auto f=std::async(find_the_answer);
    std::cout<<"the answer is "<<f.get()<<"\n";
}
```

Passing parameters

```
#include <future>
#include <iostream>
std::string copy_string(std::string const&s) {
    return s;
}

int main() {
    std::string s="hello";
    auto f=std::async(std::launch::deferred,
        copy_string,s);
    s="goodbye";
    std::cout<<f.get()<<" world!\n";
}
```

Passing parameters with `std::ref`

```
#include <future>
#include <iostream>
std::string copy_string(std::string const&s) {
    return s;
}

int main() {
    std::string s="hello";
    auto f=std::async(std::launch::deferred,
        copy_string,std::ref(s));
    s="goodbye";
    std::cout<<f.get()<<" world!\n";
}
```

Passing parameters with a lambda

```
std::string copy_string(std::string const&s) {
    return s;
}

int main() {
    std::string s="hello";
    auto f=std::async(std::launch::deferred,
        [&s](){return copy_string(s);});
    s="goodbye";
    std::cout<<f.get()<<" world!\n";
}
```

`std::async` passes exceptions

```
#include <future>
#include <iostream>
int find_the_answer() {
  throw std::runtime_error("Unable to find the answ
}
int main() {
  auto f=std::async(find_the_answer);
  try {
    std::cout<<"the answer is "<<f.get()<<"\n";
  }
  catch(std::runtime_error const& e) {
    std::cout<<"\nCaught exception: "<<e.what()<<st
  }
}
```

Promises and Tasks

Manually setting futures

- Two ways: `std::promise` and `std::packaged_task`
- `std::promise` allows you to explicitly set the value
- `std::packaged_task` is for manual task invocation, e.g. thread pools.

std::promise

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

void find_the_answer(std::promise<int>* p) {
  p->set_value(42);
}
int main() {
  std::promise<int> p;
  auto f=p.get_future();
  std::thread t(find_the_answer,&p);
  std::cout<<"the answer is "<<f.get()<<"\n";
  t.join();
}
```

std::packaged_task

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

int find_the_answer() {
  return 42;
}
int main() {
  std::packaged_task<int()> task(find_the_answer);
  auto f=task.get_future();
  std::thread t(std::move(task));
  std::cout<<"the answer is "<<f.get()<<"\n";
  t.join();
}
```

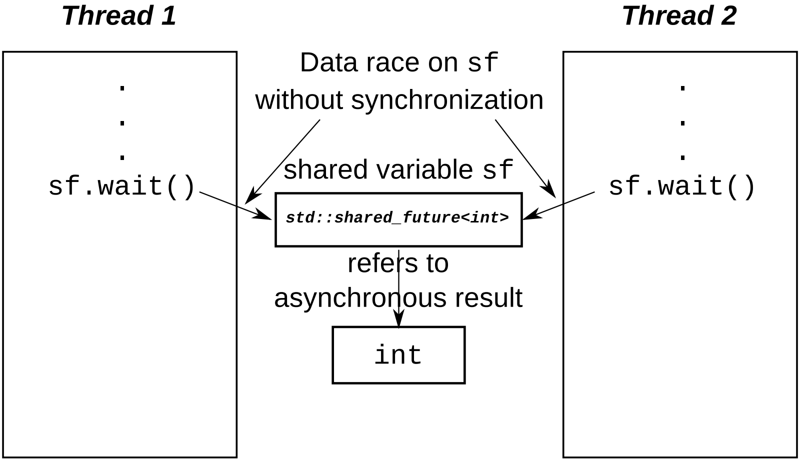
Waiting for futures from multiple threads

Use `std::shared_future<T>` rather than `std::future<T>`

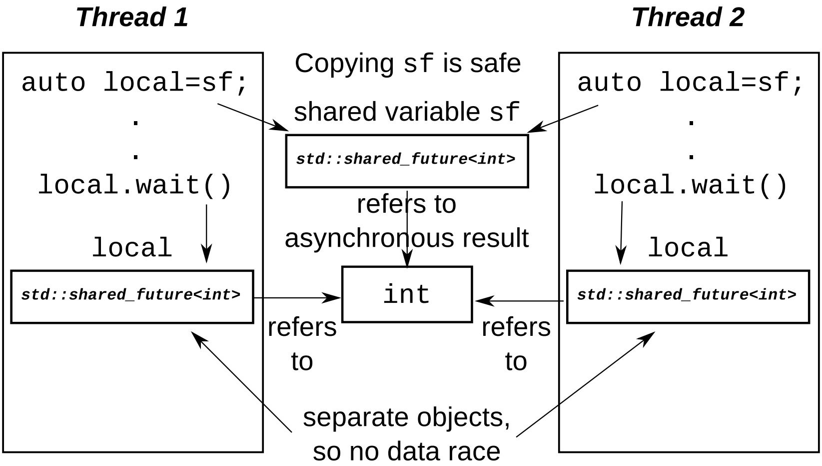
```
std::future<int> f=/*...*/;
std::shared_future<int> sf(std::move(f));

std::future<int> f2=/*...*/;
std::shared_future<int> sf2(f.share());

std::promise<int> p;
std::shared_future<int> sf3(p.get_future());
```

```
#include <future>
#include <thread>
#include <iostream>
#include <sstream>
void wait_for_notify(int id,std::shared_future<int> sf)
  std::ostringstream os;
  os<<"Thread "<<id<<" waiting\n";
  std::cout<<os.str(); os.str("");
  os<<"Thread "<<id<<" woken, val="<<sf.get()<<"\n";
  std::cout<<os.str();
}
int main() {
  std::promise<int> p;
  auto sf=p.get_future().share();
  std::thread t1(wait_for_notify,1,sf);
  std::thread t2(wait_for_notify,2,sf);
  std::cout<<"Waiting\n"; std::cin.get();
  p.set_value(42);
  t2.join(); t1.join();
}
```

std::shared_future<T> objects cannot be shared



Separate `std::shared_future<T>` objects can share state



Mutexes and Condition Variables

Lower level synchronization

- Locks and Mutexes
- Condition variables

Mutexes

C++11 has 4 mutex classes:

- `std::mutex`
- `std::recursive_mutex`
- `std::timed_mutex`
- `std::recursive_timed_mutex`

Mutex operations (I)

Mutexes have 3 basic operations, which form the `Lockable` concept:

- `m.lock()`
- `m.try_lock()`
- `m.unlock()`

Mutex operations (II)

"Timed" mutexes have 2 additional operations. A `Lockable` type that provides them satisfies the `TimedLockable` concept.

- `m.try_lock_for(duration)`
- `m.try_lock_until(time_point)`

RAII lock templates

Locking and unlocking manually is error-prone, **especially** in the face of exceptions.

C++11 provides RAII lock templates to make it easier to get things right.

- `std::lock_guard` does a simple lock and unlock
- `std::unique_lock` allows full control

```
std::mutex m;

void f(){
    m.lock();
    std::cout<<"In f()"<<std::endl;
    m.unlock();
}

int main(){
    m.lock();
    std::thread t(f);
    for(unsigned i=0;i<5;++i){
        std::cout<<"In main()"<<std::endl;
        std::this_thread::sleep_for(
            std::chrono::seconds(1));
    }
    m.unlock();
    t.join();
}
```

```
std::mutex m;

void f(){
    std::lock_guard<std::mutex> guard(m);
    std::cout<<"In f()"<<std::endl;
}

int main(){
    m.lock();
    std::thread t(f);
    for(unsigned i=0;i<5;++i){
        std::cout<<"In main()"<<std::endl;
        std::this_thread::sleep_for(
            std::chrono::seconds(1));
    }
    m.unlock();
    t.join();
}
```

```
std::mutex m;

void f(int i){
  std::unique_lock<std::mutex> guard(m);
  std::cout<<"In f("<<i<<")"<<std::endl;
  guard.unlock();
  std::this_thread::sleep_for(
    std::chrono::seconds(1));
  guard.lock();
  std::cout<<"In f("<<i<<") again"<<std::endl;
}
int main(){
  std::unique_lock<std::mutex> guard(m);
  std::thread t(f,1); std::thread t2(f,2);
  std::cout<<"In main()"<<std::endl;
  std::this_thread::sleep_for(
    std::chrono::seconds(1));
  guard.unlock();
  t2.join(); t.join();
}
```

Locking multiple mutexes

```
class account
{
    std::mutex m;
    currency_value balance;
public:
    friend void transfer(account& from,account& to,
                         currency_value amount)
    {
        std::lock_guard<std::mutex> lock_from(from.
        std::lock_guard<std::mutex> lock_to(to.m);
        from.balance -= amount;
        to.balance += amount;
    }
};
```

Locking multiple mutexes (II)

```
void transfer(account& from,account& to,
                     currency_value amount)
{
  std::lock(from.m,to.m);
  std::lock_guard<std::mutex> lock_from(
    from.m,std::adopt_lock);
  std::lock_guard<std::mutex> lock_to(
    to.m,std::adopt_lock);
  from.balance -= amount;
  to.balance += amount;
}
```

Waiting for events without futures

- Repeatedly poll in a loop (busy-wait)
- Wait using a condition variable

Waiting for an item

If all we've got is `try_pop()`, the only way to wait is to poll:

```
std::queue<my_class> the_queue;
std::mutex the_mutex;
void wait_and_pop(my_class& data) {
  for(;;){
    std::lock_guard<std::mutex> guard(the_mutex);
    if(!the_queue.empty()) {
      data=the_queue.front();
      the_queue.pop();
      return;
    }
  }
}
```

This is not ideal.

Performing a blocking wait

We want to wait for a particular condition to be true (there is an item in the queue).
This is a job for `std::condition_variable`:

```
std::condition_variable the_cv;
void wait_and_pop(my_class& data) {
    std::unique_lock<std::mutex> lk(the_mutex);
    the_cv.wait(lk,
                []()
                {return !the_queue.empty();});
    data=the_queue.front();
    the_queue.pop();
}
```

Signalling a waiting thread

To signal a waiting thread, we need to *notify* the condition variable when we push an item on the queue:

```
void push(Data const& data)
{
    {
        std::lock_guard<std::mutex> lk(the_mutex);
        the_queue.push(data);
    }
    the_cv.notify_one();
}
```

One-time Initialization

One-time initialization with `std::call_once`

```
std::unique_ptr<some_resource> resource_ptr;
std::once_flag resource_flag;

void foo()
{
  std::call_once(resource_flag,[]{
      resource_ptr.reset(new some_resource);
    });
  resource_ptr->do_something();
}
```

One-time initialization with local statics

```
void foo()
{
  static some_resource resource;
  resource.do_something();
}
```

Atomics

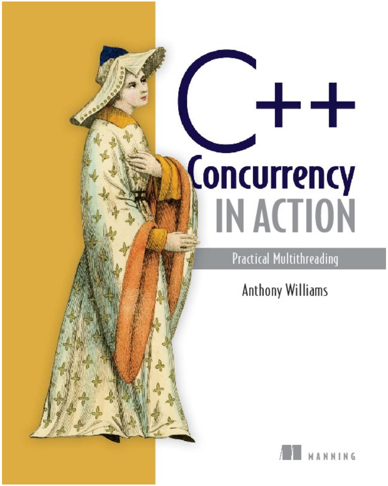
Atomic types

- Sometimes mutexes and locks are too high level
- This is where `std::atomic<T>` comes in
- Lock-free for built-in types on popular platforms
- Can use `std::atomic<POD>` — still lock-free for small structs

Just::Thread

`just::thread` provides a complete implementation of the C++11 thread library for MSVC and g++ on Windows, and g++ for Linux and MacOSX.

My Book



C++ Concurrency in Action: Practical Multithreading with the new C++ Standard.

`http://stdthread.com/book`