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`

```cpp
#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`

```cpp
#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();
}
```
```cpp
#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
}
```
#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

```cpp
#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();
}```
```cpp
#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();
}
```
#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";
}
#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";
}
#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";
}
std::string copy_string(const std::string &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";
}
#include <future>
#include <iostream>

int find_the_answer() {
    throw std::runtime_error("Unable to find the answer");
}

int main() {
    auto f = std::async(find_the_answer);
    try {
        std::cout << "the answer is " << f.get() << std::endl;
    }
    catch (std::runtime_error const& e) {
        std::cout << "Caught exception: " << e.what() << std::endl;
    }
}
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>`

```cpp
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();
}
\texttt{std::shared\_future\lt T\gt} objects cannot be shared

**Thread 1**

```
\texttt{sf.wait()}
```

**Thread 2**

```
\texttt{sf.wait()}
```

Data race on `sf` without synchronization

\texttt{std::shared\_future\lt int\gt}

refers to asynchronous result

\texttt{int}
Separate \texttt{std::shared\_future<T>} objects can share state

\begin{itemize}
  \item \texttt{auto local=sf;}
  \item \texttt{local.wait();}
\end{itemize}

\texttt{std::shared\_future<int>}

\texttt{int}

\texttt{std::shared\_future<int>}

\texttt{local.wait();}

\begin{itemize}
  \item \texttt{auto local=sf;}
  \item \texttt{local.wait();}
\end{itemize}

Copying \texttt{sf} is safe shared variable \texttt{sf}

refers to asynchronous result

refers to

separate objects, so no data race
Mutexes and Condition Variables
Lower level synchronization

- Locks and Mutexes
- Condition variables
C++11 has 4 mutex classes:

- `std::mutex`
- `std::recursive_mutex`
- `std::timed_mutex`
- `std::recursive_timed_mutex`
Mutexes have 3 basic operations, which form the **Lockable** concept:

- `m.lock()`
- `m.try_lock()`
- `m.unlock()`
“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
```cpp
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();
}
```cpp
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();
}
```
```cpp
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();
}
```
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.m);
        std::lock_guard<std::mutex> lock_to(to.m);
        from.balance -= amount;
        to.balance += amount;
    }
};
Locking multiple mutexes (II)

```cpp
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:

```cpp
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`:

```cpp
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:

```cpp
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`

```cpp
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

```cpp
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 provides a complete implementation of the C++11 thread library for MSVC and g++ on Windows, and g++ for Linux and MacOSX.
C++ Concurrency in Action: Practical Multithreading with the new C++ Standard.

http://stdthread.com/book