UNIVERSITÀ DEGLI STUDI FIRENZE

Parallel Computing

Prof. Marco Bertini

Shared memory: C++ threads

C++ multithreading

- It is possible to use Pthreads API within C++ programs.
- The C++11 standard has introduced support for multithreaded programming:
  - it allows to write programs without relying on platform-specific extensions and libraries

Thread class

- Wrap Pthreads in a class that mimics the Java Thread class
  - we do not need a Runnable interface since in C++ we have multiple inheritance…
  - use an abstract class to enforce overriding of the run() method

Thread class

```
#include <pthread.h>

class Thread {
  public:
    Thread();
    virtual ~Thread();

    int start();
    int join();
    int detach();
    pthread_t self();

    virtual void* run() = 0;

  private:
    pthread_t  tid;
    bool running;
    bool detached;
};
```

Thread class

```
#include <pthread.h>

class Thread {
  public:
    Thread();
    virtual ~Thread();

    int start();
    int join();
    int detach();
    pthread_t self();

    virtual void* run() = 0;

  private:
    pthread_t  tid;
    bool running;
    bool detached;
};
```

detaches a thread when the caller doesn't want to wait for the thread to complete.

Thread class

- Thread::Thread() : tid(0), running(false), detached(false) {}

- Thread::~Thread() {
```
    if (running && !detached) {
        pthread_detach(tid);
    }
    if (running) {
        pthread_cancel(tid);
    }
}
```

- int Thread::start() {
```
    int result = pthread_create(&tid, NULL, runThread, this);
    if (result == 0) {
        running = true;
    }
    return result;
}
```

- static void* runThread(void* arg)
```
{
    return (static_cast<Thread*>(arg))->run();
}
```

Thread class

- ```
  Thread::Thread() : tid(0),
  running(false), detached(false) {}
  ```

- ```
  Thread::~Thread() {
      if (running && !detached) {
          pthread_detach(tid);
      }
      if (running) {
          pthread_cancel(tid);
      }
  }
  ```

- ```
  int Thread::start() {
      int result =
  pthread_create(&tid, NULL,
  runThread, this);
      if (result == 0) {
          running = true;
  ```

guarantees that the internal structure is deleted, whether the thread is joined or not

- ```
  static void* runThread(void* arg)
  {
      return
  (static_cast<Thread*>(arg))-
  >run();
  }
  ```

Thread class

- ```
  Thread::Thread() : tid(0),
  running(false), detached(false) {}
  ```

- ```
  Thread::~Thread() {
      if (running && !detached) {
          pthread_detach(tid);
      }
      if (running) {
          pthread_cancel(tid);
      }
  }
  ```

- ```
  int Thread::start() {
      int result =
  pthread_create(&tid, NULL,
  runThread, this);
      if (result == 0) {
          running = true;
      }
      return result;
  }
  ```

- ```
  static void* runThread(void* arg)
  {
      return
  (static_cast<Thread*>(arg))-
  >run();
  }
  ```

Thread class

needed to let runThread to execute the run() method

```
Thread::~Thread() {
    if (running && !detached) {
        pthread_detach(tid);
    }
    if (running) {
        pthread_cancel(tid);
    }
}
```

```
int Thread::start() {
    int result =
pthread_create(&tid, NULL,
runThread, this);
    if (result == 0) {
        running = true;
    }
    return result;
}
```

```
static void* runThread(void* arg)
{
    return
(static_cast<Thread*>(arg))-
>run();
}
```

Thread class

needed to let runThread to execute the run() method

- ```
  Thread::~Thread() {
      if (running && !detached) {
          pthread_detach(tid);
      }
      if (running) {
          pthread_cancel(tid);
  ```
  ```
  }
  ```

Wraps a class method in a C function

- ```
  int Thread::start() {
      int result =
  pthread_create(&tid, NULL,
  runThread, this);
      if (result == 0) {
          running = true;
      }
      return result;
  }
  ```

- ```
  static void* runThread(void* arg)
  {
      return
  (static_cast<Thread*>(arg))-
  >run();
  }
  ```
  ```

Using the thread class

- ```
  class MyThread : public
  Thread {
      public:
      void *run() {
          for (int i = 0; i <
  5; i++) {
              printf("thread
  %lu running - %d\n",  (long
  unsigned int)self(), i+1);
              sleep(2);
          }
          printf("thread done
  %lu\n", (long unsigned
  int)self());
          return NULL;
      }
  };
  ```

- `MyThread* thread1 = new MyThread();`
- `thread1->start();`
- `thread1->join();`
  ```

Using the thread class

- ```
  class MyThread : public
  Thread {
      public:
      void *run() {
          for (int i = 0; i <
  ```

Use derived class pointer to be able to call methods specific for the subclass

- ```
                sleep(2);
          }
          printf("thread done
  %lu\n", (long unsigned
  int)self());
          return NULL;
      }
  };
  ```

- `MyThread* thread1 = new MyThread();`
- `thread1->start();`
- `thread1->join();`

Mutex class

```
class Mutex {
public:
 // just initialize to defaults
    Mutex() { pthread_mutex_init(&mutex, NULL); }
    virtual ~Mutex() { pthread_mutex_destroy(&mutex); }
    int lock() { return pthread_mutex_lock(&mutex); }
    int trylock() {
        return pthread_mutex_trylock(&mutex);
    }
    int unlock() { return pthread_mutex_unlock(&mutex); }

private:
    friend class CondVar;
    pthread_mutex_t  mutex;
};
```

Mutex class

```
class Mutex {
public:
  // just initialize to defaults
    Mutex() { pthread_mutex_init(&mutex, NULL); }
    virtual ~Mutex() { pthread_mutex_destroy(&mutex); }
    int lock() { return pthread_mutex_lock(&mutex); }
    int trylock() {
        return pthread_mutex_trylock(&mutex);
    }
    int unlock() { return pthread_mutex_unlock(&mutex); }

private:
    friend class CondVar;
    pthread_mutex_t  mutex;
};
```

If we plan to have also a class for conditional variables: this reduces the need of getter method for the Pthread mutex

Conditional variable class

```
#include "mutex.h"

class CondVar {
public:
  // just initialize to defaults
  CondVar(Mutex& mutex) : m_lock(mutex) {
     pthread_cond_init(&cond, NULL);
  }
  virtual ~CondVar() { pthread_cond_destroy(&cond); }

  int wait() {
     return  pthread_cond_wait(&cond, &(lock.mutex));
  }
  int signal() { return pthread_cond_signal(&cond); }
  int broadcast() { return pthread_cond_broadcast(&cond); }

private:
    pthread_cond_t  cond;
    Mutex& lock;
};
```

Conditional variable class

```
#include "mutex.h"

class CondVar {
public:
  // just initialize to
  CondVar(Mutex& mutex)
     pthread_cond_init(&
  }
  virtual ~CondVar() { p

  int wait() {
     return  pthread_cond_wait(&cond, &(lock.mutex));
  }
  int signal() { return pthread_cond_signal(&cond); }
  int broadcast() { return pthread_cond_broadcast(&cond); }

private:
    pthread_cond_t  cond;
    Mutex& lock;
};
```

This is why we need CondVar as friend of mutex. Otherwise we need a getter returning a reference

Boost.Thread

- More complete portable C++ classes are provided in Boost.Thread library
- `#include <boost/thread.hpp>`
- The library has been designed to follow the stye of C++11 standard thread library

C++11

Native support

- C++11 has introduced support for multithreaded programs within the language itself: there's no more need of external libraries like Pthreads.

- The C++11 standard library provides both low and high level facilities for multithread programming

Native support

- C++11 has introduced support for multithreaded programs within the language itself: there's no more need of external libraries like Pthreads.
- The C++11 standard library provides both low and high level facilities for multithread programming

Remind to compile using `-std=c++11` or `-std=c++0x`, depending on the compiler

Creating and running threads

- Use a `std::thread` object to run a function:

```
void f(int i, std::string const& s);
std::thread t(f,3,"hello");
```

- a thread object can also use:
  - classes, in this case it will execute the operator() method
  - lambda expressions
- Join a thread or detach it (without waiting for its conclusion):

- `t.join()`
- `t.detach()`

join and exceptions

- To safely join an un-detached thread try execution of code that may launch an exception followed by

```
catch(…) {
   myThreadObject.join();
   throw;
}
myThreadObject.join();
```

- or better yet use RAII

join and exceptions

Example of RAII class to manage threads

```
class ThreadGuard {
public:
    explicit ThreadGuard(std::thread& aT): t(aT) {}
    ~ThreadGuard() {
        if(t.joinable()) {
            t.join();
        }
    }
    // use new C++11 controls to eliminate default methods:
    // we do not want to allow copying of RAII object
    ThreadGuard(ThreadGuard const&)=delete;
    ThreadGuard& operator=(ThreadGuard const&)=delete;
private:
    std::thread& t;
};
```

Passing arguments

- Arguments are copied into internal thread storage also when expecting a reference
  - be careful when passing a pointer to an automatic variable !

```
void f(int i,std::string const& s);
void oops(int some_param) {
  char buffer[1024];
  sprintf(buffer, "%i",some_param);
  std::thread t(f,3,buffer);
  t.detach();
}
```

oops() may end before conversion of buffer to string is completed… undefined behavior…

```
void f(int i,std::string const& s);
void oops(int some_param) {
  char buffer[1024];
  sprintf(buffer, "%i",some_param);
  std::thread t(f,3, std::string(buffer));
  t.detach();
}
```

cast before passing to solve the issue

Passing arguments

- Arguments are copied into internal thread storage also when expecting a reference
  - be careful when passing a pointer to an automatic variable !

```
void f(int i,std::string const& s);
void oops(int some_param) {
  char buffer[1024];
  sprintf(buffer, "%i",some_param);
  std::thread t(f,3,buffer);
  t.detach();
}
```

oops() may end before conversion of buffer to string is completed… undefined behavior…

```
void f(int i,std::string const& s);
void oops(int some_param) {
  char buffer[1024];
  sprintf(buffer, "%i",some_param);
  std::thread t(f,3, std::string(buffer));
  t.detach();
}
```

cast before passing to solve the issue

If you really want to operate on a reference, perhaps to modify it, use `std::ref()`

```
std::tread t(f, 3, std::ref(myString));
```

Mutex

- `#include <mutex>`
- `std::mutex myMutex;`
- Instead of calling `lock()` on the mutex object use a C++11 RAII template object:
- `std::lock_guard<std::mutex> guard(myMutex)`

Mutex

```
std::list<int> some_list;
std::mutex some_mutex;

void add_to_list(int new_value) {
    std::lock_guard<std::mutex> guard(some_mutex);
    some_list.push_back(new_value);
}

bool list_contains(int value_to_find) {
    std::lock_guard<std::mutex> guard(some_mutex);
    return std::find(some_list.begin(),some_list.end(),value_to_find)
        != some_list.end();
}
```

Mutex

When a lock_guard object is created, it attempts to take ownership of the mutex it is given. When control leaves the scope in which the lock_guard object was created, the destructor releases the mutex.

```
s
s

void add_to_list(int new_value) {
    std::lock_guard<std::mutex> guard(some_mutex);
    some_list.push_back(new_value);
}

bool list_contains(int value_to_find) {
    std::lock_guard<std::mutex> guard(some_mutex);
    return std::find(some_list.begin(),some_list.end(),value_to_find)
        != some_list.end();
}
```

Protecting shared data

- As long as none of the member functions of an object, containing data protected with a mutex, return a pointer or reference to the protected data to their caller either via their return value or via an out parameter, the data is safe.
- But again be careful of calling alien functions that are not under control

Protecting shared data

- As long as none of the member functions of an object, containing data protected with a mutex, return a pointer or reference to the protected data to their caller either via their return value or via an out parameter, the data is safe.

Don't pass pointers and references to protected data outside the scope of the lock, whether

- by returning them from a function,
- storing them in externally visible memory,
- or passing them as arguments to user-supplied functions

Deadlock

- Instead of acquiring multiple locks on mutexes in a fixed order it is possible to lock simultaneously two or more mutexes using `std::lock()`

- `std::lock()` can be used in conjunction with `std::lock_guard<>`, asking to `lock_guard` to avoid locking the already locked mutex:

```
std::mutex m1, m2;
std::lock(m1, m2);
std::lock_guard<std::mutex> lockM1(m1, std::adopt_lock);
std::lock_guard<std::mutex> lockM2(m2, std::adopt_lock);
do_critical_operation();
```

std::unique_lock

- A std::unique_lock instance doesn't always own the mutex that it's associated with.
  - Pass std::adopt_lock as a second argument to the constructor to have the lock object manage the lock on a mutex, or pass std::defer_lock to indicate that the mutex should remain unlocked on construction.
- The lock can then be acquired later by calling lock() on the std::unique_lock object (not the mutex) or by passing the std::unique_lock object itself to std::lock().

std::unique_lock

Allows more granularity:

```
{
    std::unique_lock<std::mutex> my_lock(a_mutex);
    do_critical_work();
    my_lock.unlock();
    do_not_critical_work();
    my_lock.lock();
    do_critical_work();
    // my_lock destructor releases lock
}
```

- The lock can then be acquired later by calling `lock()` on the `std::unique_lock` object (not the mutex) or by passing the `std::unique_lock` object itself to `std::lock()`.

Condition variables

- `#include <condition_variables>`
  `std::condition_variable data_cond;`
- Use in association with a mutex
- Notify using `notify_one()`
- Wait providing the mutex and a lambda expression that checks for the expected condition: there’s no need of `while(!condition)`

Condition variables

```
std::mutex mut;
std::queue<DataType> data_queue;
std::condition_variable data_cond;

// thread adding data
DataType data = produce_data();
std::lock_guard<std::mutex> lk(mut);
data_queue.push(data);
data_cond.notify_one();

// thread consuming data
std::unique_lock<std::mutex> lk(mut);
data_cond.wait(lk, []{return !data_queue.empty();});
DataType data=data_queue.front();
data_queue.pop();
lk.unlock();
process(data)
```

Use `std::unique_lock` because:

- the wait on the condition must unlock the mutex (and thus it can not be controlled solely by `std::lock_guard`)
- it allows to explicitly unlock (we do not want `process(data)` to be synchronized)

```
// thread adding data
DataType data = produce_data();
std::lock_guard<std::mutex> lk(mut);
data_queue.push(data);
data_cond.notify_one();

// thread consuming data
std::unique_lock<std::mutex> lk(mut);
data_cond.wait(lk, []{return !data_queue.empty();});
DataType data=data_queue.front();
data_queue.pop();
lk.unlock();
process(data)
```

Thread-safe queue example

```
#include <memory> // for std::shared_ptr

template<typename T>
class threadsafe_queue {

public:
    threadsafe_queue();
    threadsafe_queue(const threadsafe_queue&);
    threadsafe_queue& operator=(const threadsafe_queue&) = delete;

    void push(T new_value);

    bool try_pop(T& value);
    std::shared_ptr<T> try_pop();

    void wait_and_pop(T& value);
    std::shared_ptr<T> wait_and_pop();

    bool empty() const;

};
```

For simplicity

Two pop variants: **`try_`** tries to pop and returns an indication of failure if queue is empty, while **`wait_`** blocks the pop. Each method has a variant: one returns data in the argument, keeping the result for errors, the other uses the return argument.

Thread-safe queue example

```
#include <memory> // for std::shared_ptr

template<typename T>
class threadsafe_queue {

public:
    threadsafe_queue();
    threadsafe_queue(const threadsafe_queue&);
    threadsafe_queue& operator=(const threadsafe_queue&) = delete;

    void push(T new_value);

    bool try_pop(T& value);
    std::shared_ptr<T> try_pop();

    void wait_and_pop(T& value);
    std::shared_ptr<T> wait_and_pop();

    bool empty() const;

};
```

For simplicity

Two pop variants: `try_` tries to pop and returns an indication of failure if queue is empty, while `wait_` blocks the pop. Each method has a variant: one returns data in the argument, keeping the result for errors, the other uses the return argument.

OK: now let's see where to really store data and manage race conditions and data access

Thread-safe queue example

```
#include <mutex>
#include <condition_variable>
#include <queue>

template<typename T>
class threadsafe_queue {

private:
    std::queue<T> data_queue;
    std::mutex mut;
    std::condition_variable data_cond;

public:
    void push(T new_value) {
        std::lock_guard<std::mutex> lk(mut);
        data_queue.push(new_value);
        data_cond.notify_one();
    }

    void wait_and_pop(T& value) {
        std::unique_lock<std::mutex> lk(mut);
        data_cond.wait(lk,[this]{return !data_queue.empty();});
        value=data_queue.front();
        data_queue.pop();
    }

};
```

Thread-safe queue example

```
#include <mutex>
#include <condition_variable>
#include <queue>

template<typename T>
class threadsafe_queue {

private:
    std::queue<T> data_queue;
    std::mutex mut;
    std::condition_variable data_cond;

public:
    void push(T new_value) {
        std::lock_guard<std::mutex> lk(mut);
        data_queue.push(new_value);
        data_cond.notify_one();
    }

    void wait_and_pop(T& value) {
        std::unique_lock<std::mutex> lk(mut);
        data_cond.wait(lk,[this]{return !data_queue.empty();});
        value=data_queue.front();
        data_queue.pop();
    }

};
```

No external synchronization required

Thread-safe queue example

```
#include <mutex>
#include <condition_variable>
#include <queue>

template<typename T>
class threadsafe_queue {

private:
    std::queue<T> data_queue;
    std::mutex mut;
    std::condition_variable data_cond;

public:
    void push(T new_value) {
        std::lock_guard<std::mutex> lk(mut);
        data_queue.push(new_value);
        data_cond.notify_one();
    }

    void wait_and_pop(T& value) {
        std::unique_lock<std::mutex> lk(mut);
        data_cond.wait(lk,[this]{return !data_queue.empty();});
        value=data_queue.front();
        data_queue.pop();
    }

};
```

No external synchronization required

Allows mutex unlock in wait

Thread-safe queue example

```
#include <mutex>
#include <condition_variable>
#include <queue>

template<typename T>
class threadsafe_queue {

private:
    std::queue<T> data_queue;
    std::mutex mut;
    std::condition_variable data_cond;

public:
    void push(T new_value) {
        std::lock_guard<std::mutex> lk(mut);
        data_queue.push(new_value);
        data_cond.notify_one();
    }

    void wait_and_pop(T& value) {
        std::unique_lock<std::mutex> lk(mut);
        data_cond.wait(lk,[this]{return !data_queue.empty();});
        value=data_queue.front();
        data_queue.pop();
    }

};
```

No external synchronization required

Allows mutex unlock in wait

automatic mutex unlock thanks to `std::unique_lock`

Thread-safe queue example

```
threadsafe_queue(threadsafe_queue const& other) {
    std::lock_guard<std::mutex> lk(other.mut);
    data_queue=other.data_queue;
}

std::shared_ptr<T> wait_and_pop() {
    std::unique_lock<std::mutex> lk(mut);
    data_cond.wait(lk,[this]{return !data_queue.empty();});
    std::shared_ptr<T> res(std::make_shared<T>(data_queue.front()));
    data_queue.pop();
    return res;
}

bool try_pop(T& value) {
    std::lock_guard<std::mutex> lk(mut);
    if(data_queue.empty)
        return false;
    value=data_queue.front();
    data_queue.pop();
    return true;
}

std::shared_ptr<T> try_pop() {
    std::lock_guard<std::mutex> lk(mut);
    if(data_queue.empty())
        return std::shared_ptr<T>();
    std::shared_ptr<T> res(std::make_shared<T>(data_queue.front()));
    data_queue.pop();
    return res;
}
```

```
bool empty() const {
    std::lock_guard<std::mutex> lk(mut);
    return data_queue.empty();
}
```

Thread-safe queue example

```
threadsafe_queue(threadsafe_queue const& other) {
    std::lock_guard<std::mutex> lk(other.mut);
    data_queue=other.data_queue;
}

std::shared_ptr<T> wait_and_pop() {
```

The method is const, but locking the mutex is a mutating operation. Therefore the mutex should be marked as mutable in its declaration. Change the previous declaration as:

`mutable std::mutex mut`

```
}

bool try_pop(T& value) {
    std::lock_guard<std::mutex> lk(mut);
    if(data_queue.empty)
        return false;
    value=data_queue.front();
    data_queue.pop();
    return true;
}

std::shared_ptr<T> try_pop() {
    std::lock_guard<std::mutex> lk(mut);
    if(data_queue.empty())
        return std::shared_ptr<T>();
    std::shared_ptr<T> res(std::make_shared<T>(data_queue.front()));
    data_queue.pop();
    return res;
}
```

```
bool empty() const {
    std::lock_guard<std::mutex> lk(mut);
    return data_queue.empty();
}
```

other is const, but locking the mutex is a mutating operation. This is another operation that need a mutable mutex.

```
threadsafe_queue(threadsafe_queue const& other) {
    std::lock_guard<std::mutex> lk(other.mut);
    data_queue=other.data_queue;
}

std::shared_ptr<T> wait_and_pop() {
```

The method is const, but locking the mutex is a mutating operation. Therefore the mutex should be marked as mutable in its declaration. Change the previous declaration as:

`mutable std::mutex mut`

```
}

bool try_pop(T& value) {
    std::lock_guard<std::mutex> lk(mut);
    if(data_queue.empty)
        return false;
    value=data_queue.front();
    data_queue.pop();
    return true;
}

std::shared_ptr<T> try_pop() {
    std::lock_guard<std::mutex> lk(mut);
    if(data_queue.empty())
        return std::shared_ptr<T>();
    std::shared_ptr<T> res(std::make_shared<T>(data_queue.front()));
    data_queue.pop();
    return res;
}
```

```
bool empty() const {
    std::lock_guard<std::mutex> lk(mut);
    return data_queue.empty();
}
```

Using the thread-safe queue

```
threadsafe_queue<data_chunk> data_queue;

void data_preparation_thread() {
    while(more_data_to_prepare()) {
        data_chunk const data=prepare_data();
        data_queue.push(data);
    }
}

void data_processing_thread() {
    while(true) {
        data_chunk data;
        data_queue.wait_and_pop(data);
        process(data);
        if(is_last_chunk(data))
            break;
    }
}
```

Atomic types

- C++ provides many atomic types in <atomic>, that provides synchronization “under the hood” in their implementation, e.g.:
- `std::atomic<int>` , is also available as `atomic_int` type
- The standard atomic types are not copyable or assignable in the conventional sense, in that they have no copy constructors or copy assignment operators.
  They do, however, support assignment from and implicit conversion to the corresponding built-in types as well as direct `load()` and `store()` member functions, `exchange()`, `compare_exchange_weak()`, and `compare_exchange_strong()`
- They have many operators and support for pointer operations

Atomic types

Compare/exchange operation is the cornerstone of programming with atomic types; it compares the value of the atomic variable with a supplied expected value and stores the supplied desired value if they're equal. If the values aren't equal, the expected value is updated with the actual value of the atomic variable.

- The standard atomic types are not copyable or assignable in the conventional sense, in that they have no copy constructors or copy assignment operators.
  They do, however, support assignment from and implicit conversion to the corresponding built-in types as well as direct `load()` and `store()` member functions, `exchange()`, `compare_exchange_weak()`, and `compare_exchange_strong()`
- They have many operators and support for pointer operations

Future and async

- `std::async` provides facilities for a higher-level parallelism than `std::thread`
  - returns more easily results from threads (no need to use pointer args)
  - allows to defer thread launch
  - executes asynchronously

Future and async

```
void accumulate_block_worker(int* data, size_t count, int* result) {
  *result = std::accumulate(data, data + count, 0);
}

void use_worker_in_std_thread() {
  std::vector<int> v{1, 2, 3, 4, 5, 6, 7, 8};
  int result;
  std::thread worker(accumulate_block_worker,
                     v.data(), v.size(), &result);
  worker.join();
  std::cout << "use_worker_in_std_thread computed " << result << "\n";
}
```

Future and async

```
int accumulate_block_worker_ret(int* data, size_t count) {
  return std::accumulate(data, data + count, 0);
}

void use_worker_in_std_async() {
  std::vector<int> v{1, 2, 3, 4, 5, 6, 7, 8};
  std::future<int> fut = std::async(
      std::launch::async, accumulate_block_worker_ret, v.data(),
v.size());
  std::cout << "use_worker_in_std_async computed " << fut.get() <<
"\n";
}
```

Future and async

```
int accumulate_block_worker_ret(int* data, size_t count) {
  return std::accumulate(data, data + count, 0);
}

void use_worker_in_std_async() {
  std::vector<int> v{1, 2, 3, 4, 5, 6, 7, 8};
  std::future<int> fut = std::async(
      std::launch::async, accumulate_block_worker_ret, v.data(),
v.size());
  std::cout << "use_worker_in_std_async computed " << fut.get() <<
"\n";
}
```

Use explicitly this execution politics (the default allows also deferral)

Future

- std::future decouples the task from the result

  - bonus: you can pass the future somewhere else, and it encapsulates both the thread to wait on and the result you'll end up with.

    Useful in the scenario in which we want to launch tasks in one place but collect results in some other place.

```
using int_futures = std::vector<std::future<int>>;

int_futures launch_split_workers_with_std_async(std::vector<int>& v) {
  int_futures futures;
  futures.push_back(std::async(std::launch::async,
accumulate_block_worker_ret,
                               v.data(), v.size() / 2));
  futures.push_back(std::async(std::launch::async,
accumulate_block_worker_ret,
                               v.data() + v.size() / 2, v.size() /
2));
  return futures;
}


...

{
  // Usage
  std::vector<int> v{1, 2, 3, 4, 5, 6, 7, 8};
  int_futures futures = launch_split_workers_with_std_async(v);
  std::cout << "results from launch_split_workers_with_std_async: "
            << futures[0].get() << " and " << futures[1].get() <<
"\n";
}
```

Future and time out

- It is possible to time out on futures, so to avoid to be blocked on long computations
  - instead joining threads does not allow this. We need to create a control structure with condition variables.

Future and time out

```
int accumulate_block_worker_ret(int* data, size_t count) {
  std::this_thread::sleep_for(std::chrono::seconds(3));
  return std::accumulate(data, data + count, 0);
}

int main(int argc, const char** argv) {
  std::vector<int> v{1, 2, 3, 4, 5, 6, 7, 8};
  std::future<int> fut = std::async(
      std::launch::async, accumulate_block_worker_ret, v.data(),
v.size());
  while (fut.wait_for(std::chrono::seconds(1)) !=
std::future_status::ready) {
    std::cout << "... still not ready\n";
  }
  std::cout << "use_worker_in_std_async computed " << fut.get() <<
"\n";

  return 0;
}
```

Exceptions and threads

- The C++ standard states, "`~thread()`, if `joinable()`, calls `std::terminate()`".
  So trying to catch the exception of a thread in another thread won't help:

```
try {
  std::thread worker(accumulate_block_worker, v.data(),
v.size(), &result);
  worker.join();
  std::cout << "use_worker_in_std_thread computed " << result
<< "\n";
} catch (const std::runtime_error& error) {
  std::cout << "caught an error: " << error.what() << "\n";
}
```

  Results in:

```
terminate called after throwing an instance of
'std::runtime_error'
  what():  something broke
Aborted (core dumped)
```

Solution:

Use `std::future`, since it propagates exceptions:

```
int accumulate_block_worker_ret(int* data, size_t count) {
  throw std::runtime_error("something broke");
  return std::accumulate(data, data + count, 0);
}


...


{
  std::vector<int> v{1, 2, 3, 4, 5, 6, 7, 8};
  try {
    std::future<int> fut = std::async(
        std::launch::async, accumulate_block_worker_ret, v.data(),
v.size());
    std::cout << "use_worker_in_std_async computed " << fut.get() <<
"\n";
  } catch (const std::runtime_error& error) {
    std::cout << "caught an error: " << error.what() << "\n";
  }
}
```

Future and deferred async

- The deferred policy means that the task will run lazily on the calling thread only when `get()` is called on the future it returns.
- The default `std::async` let the runtime choose either to execute async or deferred, but the code to manage both cases may become complicated
- Good practice: always explicitly execute with `std::launch::async`

Future and deferred async

- The deferred policy means that the task will run lazily on the calling thread only when `get()` is called on the future it returns.

Scott Meyers suggests to use this wrapper to ensure to always launch async:

```
template <typename F, typename... Ts>
inline auto reallyAsync(F&& f, Ts&&... params) {
  return std::async(std::launch::async, std::forward<F>(f),
                    std::forward<Ts>(params)...);
}
```

- Good practice: always explicitly execute with `std::launch::async`

Future and deferred async

- The deferred policy means that the task will run lazily on the calling thread only when `get()` is called on the future it returns

forwards the argument to another function with the value category (e.g. lvalue, rvalue) it had when passed to the calling function.

```
template <typename F, typename... Ts>
inline auto reallyAsync(F&& f, Ts&&... params) {
  return std::async(std::launch::async, std::forward<F>(f),
                    std::forward<Ts>(params)...);
}
```

- Good practice: always explicitly execute with `std::launch::async`

STL Thread safety

- All const member functions can be called concurrently by different threads on the same container. In addition, the member functions `begin()`, `end()`, `rbegin()`, `rend()`, `front()`, `back()`, `data()`, `find()`, `lower_bound()`, `upper_bound()`, `equal_range()`, `at()`, and, except in associative containers, `operator[]`, behave as const for the purposes of thread safety.

- Any member function that invalidates iterators, such as `vector::push_back` or `set::erase`, requires synchronization with every thread that accesses any iterator, even the ones that aren't invalidated.

- Different elements in the same container can be modified concurrently by different threads, except for the elements of `std::vector<bool>`

STL Thread safety

Basically reading from a container from multiple threads is fine, and modifying elements that are already in the container is fine (as long as they are different elements).

But:

- having two threads inserting into a vector/list is not thread-safe: they are modifying the vector/list itself - not existing separate elements.
- one thread erasing and other walking to access the same element is not thread safe

Container operations that invalidate any iterators modify the container and cannot be executed concurrently with any operations on existing iterators even if those iterators are not invalidated.

Books

- C++ Concurrency in action: practical multithreading, Anthony Williams, Manning - Chapt. 2-5