C++ Threading Support

- Before C++11, language specification didn’t say anything about multithreaded programs
- No guarantee of portability of multithreaded programs across multiple platforms
- Rely on OS threading support (e.g. POSIX pthreads, Windows threads)
- For more advanced multithreading programs, e.g. lock-free data structures
  - Often need assembly language snippets and other language tricks to make code behave properly
- Not ideal, but most definitely usable
C++ Threading Support (2)

- C++11 adds language support for multithreading
  - Specification is now written from the perspective of multithreaded programs
- Includes facilities for creating basic multithreaded programs
  - Thread, mutex, condition variable classes
  - Helper classes and functions for managing them
- Also, facilities for more advanced multithreaded programming
  - Atomic data types with various memory-ordering guarantees
  - Other abstractions, e.g. async. tasks and futures
The `std::thread` class is a wrapper around an operating-system thread:

```cpp
#include <thread>
```

- Constructor takes a function and a set of arguments:
  - A new thread is started, then the thread calls the function with the arguments.
- Thread terminates when the function returns.
Constructor is a function template:

```cpp
template <class Fn, class... Args>
explicit thread(Fn&& fn, Args&&... args)
```

- `Fn` can be a pointer to a function, or any object that implements `()` operator
  - i.e. anything that can be called like a function
- `Args` is a parameter pack
  - C++11 adds variadic templates to the language
  - Allows the thread constructor to take as many arguments as `Fn` requires
Constructor is a function template:

```cpp
template <class Fn, class... Args>
explicit thread(Fn&& fn, Args&&... args)
```

Constructor is marked `explicit` to prevent implicit conversions from functions to threads.

- Cases where the function `Fn` takes no arguments
  - e.g. without `explicit`, this would start a thread:
    ```cpp
    thread t = rand;
    ```

- All arguments to thread constructor must also be `move-constructible`
  - i.e. they must have either a move constructor or a copy constructor
std::thread also has a default constructor

The thread object doesn’t represent a thread of execution

You can’t do much with the object except assign another thread to it (more on this in a moment)
Only one `std::thread` object may represent a given OS-level thread of execution.

`std::thread` defines a move constructor:

```cpp
thread(thread &&x)
```

This allows us to create functions that return `thread` objects, etc.

`std::thread` does not define a copy constructor.
thread Assignment Operator

- Only one `std::thread` object may represent a given OS-level thread of execution.
- Similarly, `std::thread` defines a move-assignment operator.
  - Ensures that only one thread object represents each thread of execution.
- Example:
  ```cpp
  thread t1(f, a, b, c); // Calls f(a, b, c)
  thread t2;
  
t2 = t1; // Contents of t1 are moved to t2!
  // After assignment, t1 is empty!
  ```
Joining and Detaching Threads

- Can find out when a thread terminates by calling its `join()` member-function
  - `join()` doesn't return until the thread of execution terminates
- Also, threads hold some system resources that *must* be reclaimed at termination
  - A stack, exit status, etc.
- Calling `join()` causes these resources to be reclaimed
- If a program doesn’t care when a thread terminates, can call `detach()` on it
  - Cannot call `join()` on the thread later
  - The thread’s resources are cleaned up automatically
Joinable Threads

- Can only call `join()` on threads that are `joinable`
  - The `joinable()` method reports if it is joinable
- A thread object is not joinable if:
  - The thread object was initialized via the default constructor (and no other thread object was moved into it)
  - The thread object's contents were moved into another thread object
  - Either `join()` or `detach()` has been called on the thread object
- If a thread object is joinable when its destructor is called, your program will be terminated!
Aside: Removing Copy Constructors

- Before C++11, the usual way to remove special member functions was to make them private
  - e.g. copy constructor, assignment operator, destructor, new operator, etc.

```cpp
class NotCopyable {
    NotCopyable(const NotCopyable &){}

public:
    ...;
};
```

- Attempts to copy objects of this type produce a compile error (but one about private members)
Aside: Removing Copy Constructors (2)

C++11 introduces a new way to remove special functions:

```cpp
class NotCopyable {
public:
    NotCopyable(const NotCopyable &) = delete;
    ...
};
```
What happens if a thread’s function throws?

```c++
void f() {
    throw exception();
}
```

// Start a thread running f
thread t(f);

In C++, if an exception is unhandled, the runtime calls the `std::terminate()` function

- Default impl calls `abort()` - terminates the process
- (can replace default `terminate()` handler too)

If a thread’s function throws an exception, the default behavior is to kill your entire program!
The Current Thread

- C++ also has a `std::this_thread` namespace
  - Provides several helper functions for performing operations on the current thread
- Suspend the thread for a period of time:
  ```cpp
  void this_thread::sleep_for(
      const std::chrono::duration &duration)
  ```
  - Sleep for a certain duration
- Sleep until a specific point in time:
  ```cpp
  void this_thread::sleep_until(
      const std::chrono::time_point &time)
  ```
The Current Thread (2)

- Can also yield the processor to another thread
  
  ```cpp
  void this_thread::yield()
  ```

  - Allows other threads of execution to be scheduled on the processor

  - Can be helpful if the current thread has more work to do, but wants to give other threads a chance to progress as well
C++11 Standard Library also includes mutexes and condition variables

**Mutexes** are mutual-exclusion locks
- Can be used to ensure exclusive access to resources

**Condition variables** can be used to passively wait for some condition to become true
- Can also timeout, in case condition doesn’t become true within a time limit
- Another thread can notify threads waiting on a condition variable, to wake them up
Mutexes

- C++ provides several different kinds of mutexes
  ```
  #include <mutex>
  ```

- `std::mutex`
  - `void lock()` – locks the mutex, blocking if it is not available
  - `bool try_lock()` – attempts to lock the mutex, but returns immediately if it is not available
    - Returns `true` if lock acquired; `false` otherwise
  - `void unlock()` – unlocks a previously locked mutex

- Mutex cannot be assigned or copied (makes sense)
- This class **does not** support multiple recursive lock requests from the same thread!
  - May deadlock, or may cause an exception
Mutexes (2)

- `std::recursive_mutex`
  - Same set of member functions as `std::mutex`
  - Supports multiple recursive lock requests from the same thread
    - Implementation is slightly more complex than `std::mutex`, to detect recursive lock requests
  - Ownership of mutex extends from first `lock()` call to last matching `unlock()` call
    - Intervening calls to `lock()` and `unlock()` don’t affect state of mutex
Mutexes (3)

- Also `timed_mutex` and `recursive_timed_mutex`
- These classes support lock-requests with a timeout
  - `bool try_lock_for(
    const std::chrono::duration &duration)
  
  Try to acquire lock for a certain duration, then give up`
  - `bool try_lock_until(
    const std::chrono::time_point &timeout_time)
  
  Try to acquire lock until a specific point in time, then give up`
Managing Mutexes

- Mutexes can present some management issues
- Example: thread-safe FIFO

```cpp
template<typename T, int maxElems>
class ThreadSafeFIFO {
  typedef struct list_elem {
    T *data;
    list_elem *next;
  };
  mutex m; // To guard FIFO internals
  int numElems; // How many elems in FIFO?
  list_elem *head, *tail; // Linked list of elems

public:
  ...
};
```
Managing Mutexes (2)

- Example: thread-safe FIFO (cont)
  
  template<typename T, int maxElems>
  class ThreadSafeFIFO {
    ...
    void put(T* data) {
      m.lock();
      if (numElems >= maxElems)
        throw fifo_full();
      list_elem *e = new list_elem();
      ... // Add element to linked list
      m.unlock();
    }
  };

  Problems?

  Need to unlock mutex before throwing!

  If this allocation fails, std::bad_alloc will be thrown! (same issue)
C++ also provides classes for managing mutex locks based on variable scoping.

Recall: when an object goes out of scope, its destructor is called.

- Allows object to release any resources it holds
- Allows us to implement the “Resource Acquisition Is Initialization” (RAII) pattern
- Can use this to implement “smart pointers” which automatically delete the memory they point to

Idea: Why not manage mutexes in the same way?

- A wrapper class that represents an acquired lock
- When the object goes out of scope, it calls `unlock()`
Managing Mutexes (4)

- The `std::lock_guard<class Mutex>` template manages a mutex-lock
  - Also in `<mutex>` header
  - Specify the kind of mutex being managed in the template parameter
- `lock_guard(Mutex &m)` constructor acquires a lock on `m`
- When `lock_guard` object passes out of scope, `m.unlock()` is called automatically
- Can use this class to implement lock mgmt that is safe in the context of exceptions
Managing Mutexes (5)

- Our thread-safe FIFO, revisited:
  ```cpp
template<typename T, int maxElems>
class ThreadSafeFIFO {
...
  void put(T* data) {
    lock_guard<mutex> lock(m);
    if (numElems >= maxElems)
      throw fifo_full();
    list_elem *e = new list_elem();
    ... // Add element to linked list
  }
};
```
  - lock goes out of scope, calls `m.unlock()`

- Now, when `lock` goes out of scope, `m` is unlocked automatically (even when exceptions are thrown)
Managing Mutexes (6)

- `std::lock_guard` is for very simple lock/unlock scenarios
- The `std::unique_lock` wrapper is a much more general purpose lock wrapper
  - Exposes most mutex operations directly on the lock wrapper
  - Also ensures that the mutex will be released when the wrapper goes out of scope
Next Time

- Threads are kind of finicky...
  - C++ has some helpful higher-level abstractions that make writing concurrent code a little easier

- Next time, continue exploring C++ multithreading support