# Lock Guard Solutions

# Exception in critical section

- Explain what happens if an exception is thrown in a critical section
  - The program will immediately leave the current scope and start executing an exception handler
  - If the critical section is protected by calling lock() on std::mutex, the unlock() call will never be executed
  - All threads which are waiting to acquire a lock on the mutex will be blocked
  - All threads which are waiting or joined on these threads will be blocked
  - Usually, the entire program is deadlocked

# Exception in critical section

- What approaches can programmers use to manage this situation?
  - C++ provides a number of "wrapper" classes that use the RAII idiom to manage mutexes
  - Examples: lock_guard, scope_guard
  - These take advantage of the fact that, when an exception is thrown, the destructors are called for every object in scope
  - These wrapper classes lock the mutex in their constructor and unlock it in their destructor
  - This guarantees that the mutex will always be unlocked if the function returns

# Exception in critical section

- Suggest some situations other than exceptions being thrown in which these approaches could be useful
  - Thread function with multiple return paths ("return" statements)
  - Loop which locks a mutex and has "break" or "continue" statements
  - The programmer keeps forgetting to call unlock(!)

# std::lock_guard

- Rewrite the unscramble program from the last exercise to use a lock_guard instead of locking and unlocking a mutex directly
- Do you notice any difference in running time between the two versions? Why might this be the case?
  - On my system, the new version took more then 20 times as long to execute
  - In the first version, the mutex is unlocked immediately after the critical section
  - In the new version, the mutex is not unlocked until the end of the loop, including the non-critical sleep statement
  - This prevents any other threads from running while the current thread is sleeping

# std::lock_guard

- Rewrite the "unscramble with exception" program from the last exercise to use an std::lock_guard instead of locking and unlocking a mutex directly
- What happens when you run the program? Explain your results.
  - The program appears to run normally, except that each thread iteration's output is followed by the exception handler's output
  - When the exception is thrown, the destructors are called for all objects in scope and the program jumps into the catch handler
  - The lock_guard destructor will unlock the mutex
  - One of the threads which are waiting for the lock will acquire the lock and be able to run
  - No threads are blocked

# std::lock_guard

- Suggest one feature that could usefully be added to std::lock_guard
  - A member function to unlock the mutex
  - This would give programmers more control over when the mutex is unlocked, while still having the fall-back of release on destruction
  - It would avoid the problem of blocking other threads while executing code after the critical region