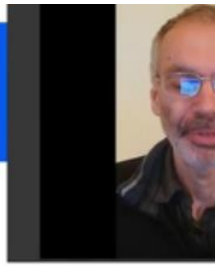
Torn Writes and Reads



- A "torn" write
  - Can occur when writing data requires multiple operations
  - Another writing thread interleaves between the write operations

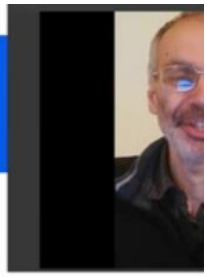
Threads A and B write to a memory location	0x????
Thread A starts writing 0x1234	0x12??
Thread B interleaves and writes 0x4567	0x4567
Thread A continues writing 0x1234	0x4534
The memory location contains part of Thread A's number and part of Thread B's number	0x4534

Torn Writes and Reads

- A "torn" read
  - Can occur when reading and writing data requires multiple opera
  - A writing thread interleaves between the read operations

Threads A reads from a memory location	0x4321
Thread A starts reading 0x4321	0x4321
Thread B interleaves and writes 0x9876	0x9876
Thread A continues reading	0x9876
Thread A has read 0x4376	
Thread A has read part of the original value and part of the new value	

Improper Construction



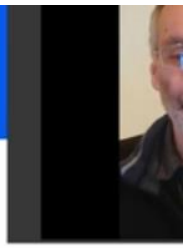
- Thread A creates a shared object
  - It calls the class's constructor
- Thread B interleaves
  - Thread B also calls the class's constructor, at the same memory location
- We have a torn write
  - The object will be initialized with a mixture of values from each thread

Improper Construction

- Partially constructed object

```
// Vector of polymorphic objects
std::vector<Base *> vec;

// The Base constructor adds the object to the vector
Base::Base() {
  vec.push_back(this);
}

// The Derived constructor calls the Base constructor first
// Then it initializes the Derived members
Derived::Derived() : Base(), ...
```

Improper Construction



- Thread A calls Derived's constructor
  - Derived's constructor calls Base's constructor
  - Base's constructor pushes the Base part of the object onto the vector
- Thread B interleaves
  - Thread B accesses the element in vec
  - The Derived constructor has not completed
  - Thread B will see a partially constructed object
- Thread A initializes the Derived part of the object

Improper Destruction

- Destructor of reference-counted object

```
*ref_count--;                // (1)
if (*ref_count == 0) {       // (2)
    delete p;
    delete ref_count;
}
```

- Thread A decrements the counter
  - Thread B interleaves and decrements the counter
  - Thread A checks the counter and may release the members
  - Thread B checks the counter and may release the members

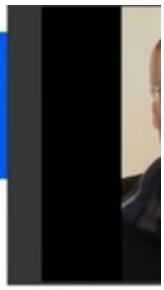
Improper Destruction

- If ref_count was initially 2
  - p and ref_count could be deleted twice
- If ref_count was initially 1
  - p and ref_count may not be deleted at all

Managing Data Races

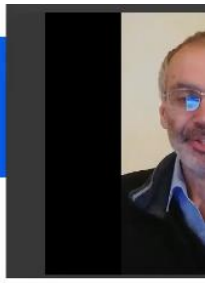
- There are no "benign" data races
  - All data races in C++ are potentially dangerous
- Can be very difficult to detect and replicate
  - Intermittent errors
  - Sensitive to environment
  - Often dependent on timing coincidences or system load
- The only good solution is to prevent data races from occurring

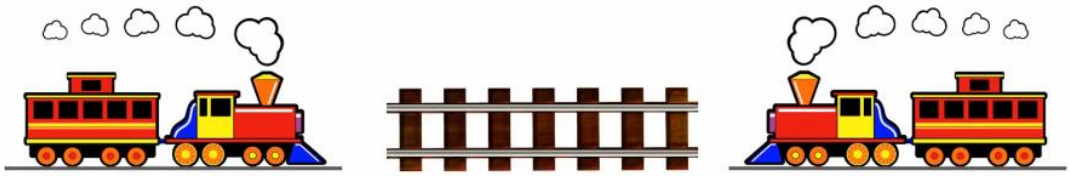
Shared Data



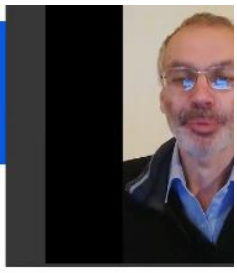
- Avoid sharing data between different threads
- If unavoidable, synchronize the threads
  - Impose an ordering on how the threads access the shared data
- This has substantial costs
  - Increased execution time
  - Increased program complexity

Single-track Railway





Single-track Railway



Single-track Railway

- The track can only be used by one train at a time
- When a train approaches this section of track:
  - If there is another train in the section, wait
  - Proceed only when there is no other train

Critical Section

- A region of code
- Must only be executed by one thread at a time
- Usually when accessing a shared resource
  - Shared data, network connection, hardware device
- The thread "enters" the critical section
  - It starts executing the code in the critical section
- The thread "leaves" the critical section
  - It has executed all the code in the critical section

Locking Protocol

- One thread can enter the critical section
  - All the other threads are "locked out"
  - Only this thread can execute the code in the critical section
- The thread leaves the critical section
  - One of the other threads can now enter it

Mutex

- MUTual EXclusion object
- We can use a mutex to implement locking
- A mutex has two states
  - "locked"
  - "unlocked"

Mutual Exclusion

- Exclusion
  - The mutex is used to exclude threads from the critical section
- Mutual
  - The threads agree to respect the mutex
- Locking
  - If the mutex is unlocked, a thread can enter the critical section
  - If the mutex is locked, no thread can enter until it becomes unlocked
  - A thread locks the mutex when it enters the critical section
  - A thread unlocks the mutex when it leaves the critical section

Thread Synchronization with Mutex

- Some threads A, B, C, ... wish to enter a critical section
  - Thread A locks the mutex
  - Thread A enters the critical section
  - Threads B, C, ... wait until they can lock the mutex
  - Thread A leaves the critical section
  - Thread A unlocks the mutex
  - One of threads B, C, ... can now lock the mutex and enter the critical section

Thread Synchronization with Mutex

- The threads are synchronized
  - They cannot interleave when they execute in the critical section
  - There is no data race
- Unlocking a mutex "publishes" any changes
  - Thread A modifies shared data
  - The new value is now available to other threads
  - It will be seen by the next thread which accesses the shared data

Acquire-Release Semantics

- A thread locks a mutex
  - It "acquires" exclusive access to the critical section
- The thread unlocks the mutex
  - It "releases" exclusive access to the critical section
  - It also "releases" the result of any modifications
  - The next thread that locks the mutex will acquire these results
- These acquire-release semantics impose ordering on the threads
  - There is no data race
  - The shared data is always in a consistent state

std::mutex Class

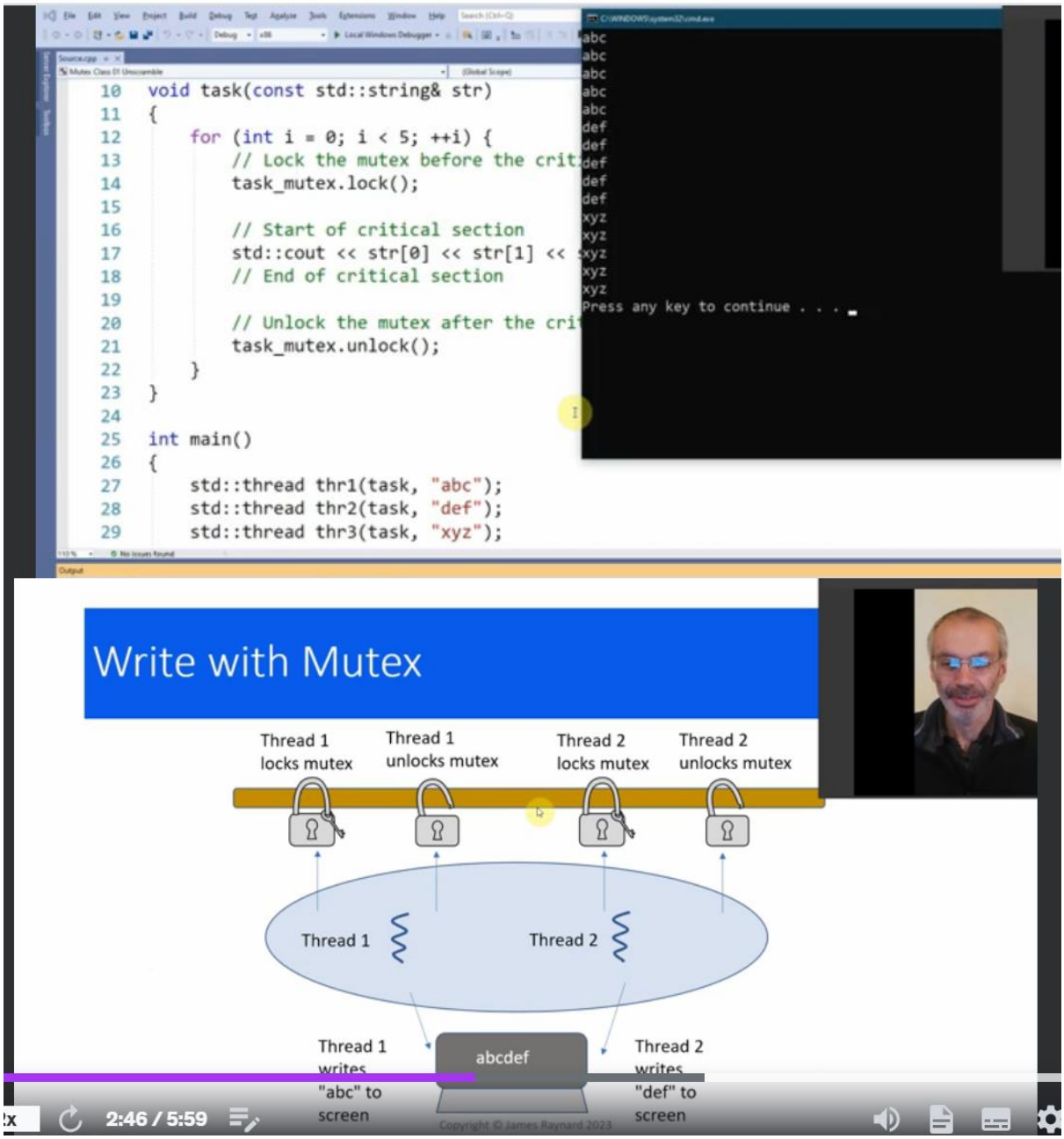
- The C++ Standard Library provides an std::mutex class
  - Defined in <mutex>
- A mutex object must be visible in all task functions which uses it
- It must also be defined outside the task functions
  - Global/static variable with global task function
  - Class data member with member task function
  - Variable captured by reference with lambda expressions

std::mutex Interface

- Three main member functions:
- lock()
  - Tries to lock the mutex
  - If not successful, waits until it locks the mutex
- try_lock()
  - Tries to lock the mutex
  - Returns immediately if not successful
- unlock()
  - Releases the lock on the mutex

Rewrite using std::mutex

```
// Global mutex object
std::mutex task_mutex;

void task(const std::string& str)
{
   for (int i = 0; i < 5; ++i) {
      // Lock the mutex before the critical section
      task_mutex.lock();

      // The thread has exclusive access to the critical section
      std::cout << str[0] << str[1] << str[2] << std::endl;

      // Unlock the mutex after the critical section
      task_mutex.unlock();
   }
}
```


10 void task(const std::string& str)
11 {
12 for (int i = 0; i < 5; ++i) {
13 // Lock the mutex before the crit
14 task_mutex.lock();
15
16 // Start of critical section
17 std::cout << str[0] << str[1] << s
18 // End of critical section
19
20 // Unlock the mutex after the cri
21 task_mutex.unlock();
22 }
23 }
24
25 int main()
26 {
27 std::thread thr1(task, "abc");
28 std::thread thr2(task, "def");
29 std::thread thr3(task, "xyz");
abc
abc
abc
abc
abc
def
def
def
def
def
def
xyz
xyz
xyz
xyz
xyz
Press any key to continue . . .
Write with Mutex
Thread 1 locks mutex
Thread 1 unlocks mutex
Thread 2 locks mutex
Thread 2 unlocks mutex
Thread 1
Thread 2
Thread 1 writes "abc" to screen
abcdef
Thread 2 writes "def" to screen
Copyright © James Raynard 2023
2:46 / 5:59

Output

- The output is no longer scrambled up
- The accesses to the critical section are synchronized
  - This prevents the threads from interfering with each other

```
abc
abc
abc
abc
abc
def
def
```

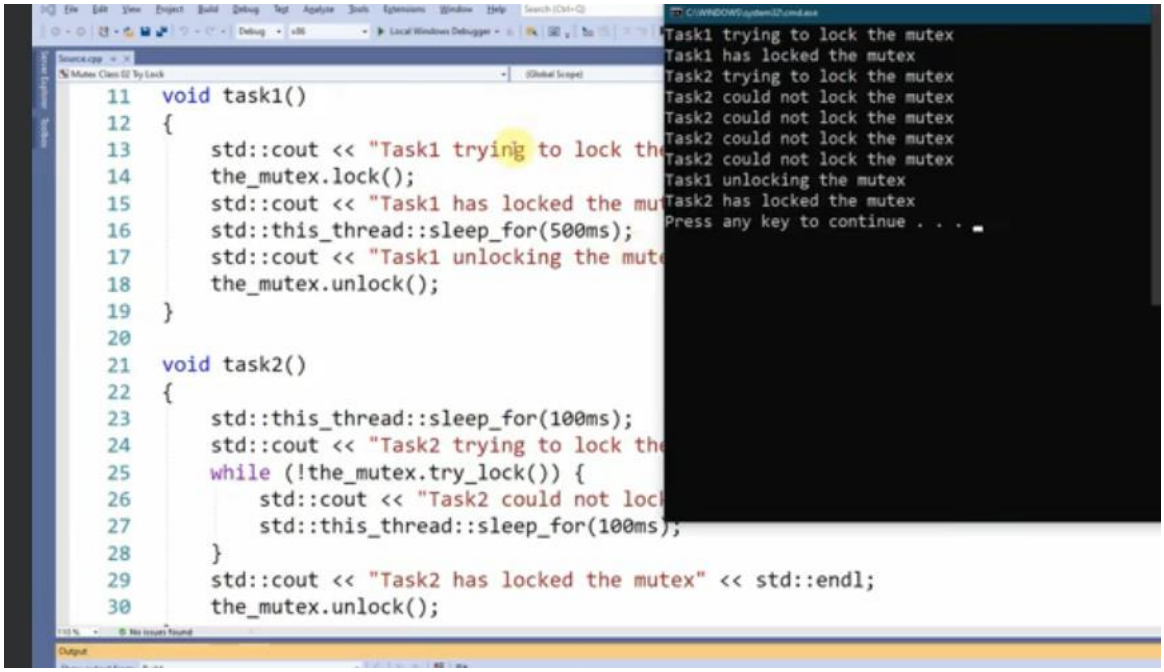
std::mutex::try_lock()

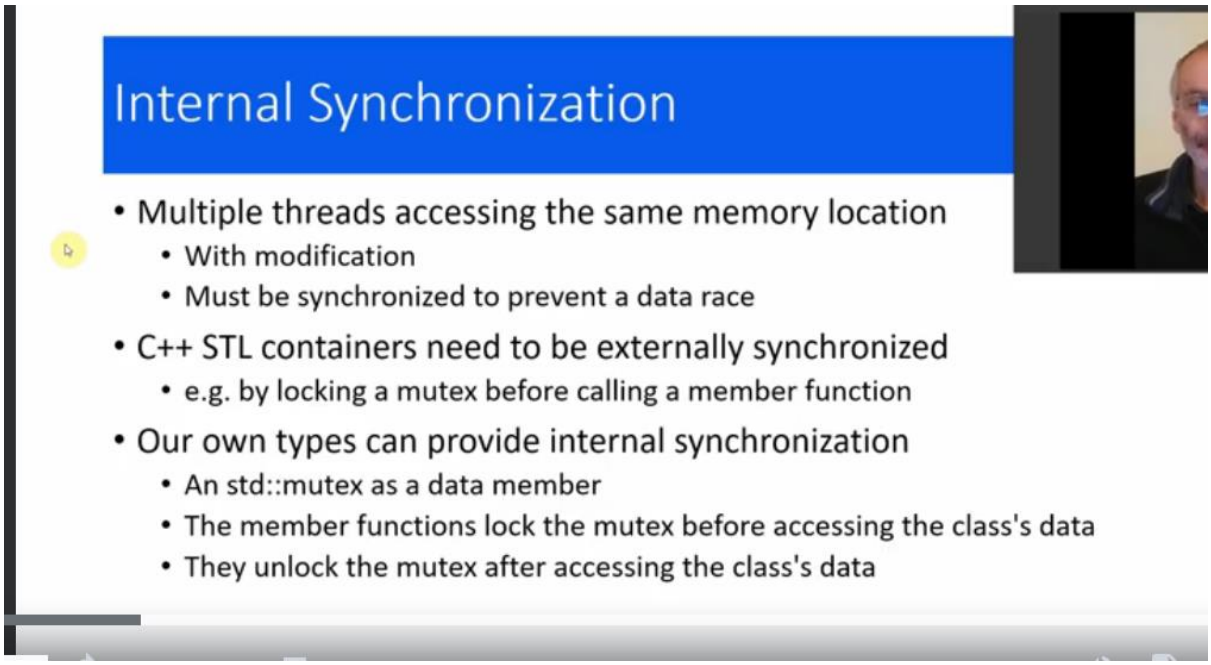
- try_lock() returns immediately
  - Returns true if it locked the mutex
  - Returns false if it could not lock the mutex

std::mutex::try_lock()

- Usually called in a loop

```
// Keep trying to get the lock
while (!the_mutex.try_lock()) {

  // Could not lock the mutex
  // Try again later
  std::this_thread::sleep_for(100ms);
}

// Finally locked the mutex
// Can now execute in the critical section
```

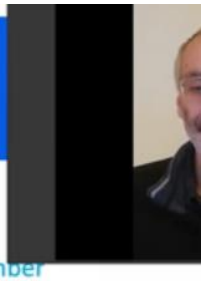

11 void task1()
12 {
13 std::cout << "Task1 trying to lock the
14 the_mutex.lock();
15 std::cout << "Task1 has locked the mut
16 std::this_thread::sleep_for(500ms);
17 std::cout << "Task1 unlocking the mute
18 the_mutex.unlock();
19 }
20
21 void task2()
22 {
23 std::this_thread::sleep_for(100ms);
24 std::cout << "Task2 trying to lock the
25 while (!the_mutex.try_lock()) {
26 std::cout << "Task2 could not lock
27 std::this_thread::sleep_for(100ms);
28 }
29 std::cout << "Task2 has locked the mutex" << std::endl;
30 the_mutex.unlock();
Task1 trying to lock the mutex
Task1 has locked the mutex
Task2 trying to lock the mutex
Task2 could not lock the mutex
Task2 could not lock the mutex
Task2 could not lock the mutex
Task1 unlocking the mutex
Task2 has locked the mutex
Press any key to continue . . .


Internal Synchronization
• Multiple threads accessing the same memory location
• With modification
• Must be synchronized to prevent a data race
• C++ STL containers need to be externally synchronized
• e.g. by locking a mutex before calling a member function
• Our own types can provide internal synchronization
• An std::mutex as a data member
• The member functions lock the mutex before accessing the class's data
• They unlock the mutex after accessing the class's data

Wrapper for std::vector

- std::vector acts as a memory location
  - We may need to lock a mutex before calling its member function
- Alternatively, we could write an internally synchronized wrapper for it
- A class which
  - Has an std::vector data member
  - Has an std::mutex data member
  - Member functions which lock the mutex before accessing the std::vector
  - Then unlock the mutex after accessing it
- An internally synchronized class

Wrapper for std::vector



```
// Very simplistic thread-safe vector class
class Vector {
  std::mutex mut;                    // Mutex as private class data member
  std::vector<int> vec;              // Shared data - mutex protects access to it
 public:
  void push_back(const int& i) {
    mut.lock();                      // Lock the mutex
    vec.push_back(i);                // Critical section
    mut.unlock();                    // Unlock the mutex
  }
};
```

Exception Thrown in Critical Section

```
try {
    task_mutex.lock();          // Lock the mutex before the critical section

    // Critical section throws an exception
    ...

    task_mutex.unlock();        // Never gets called
}
catch (std::exception& e) {
    ...
}
```

- The mutex will be left locked

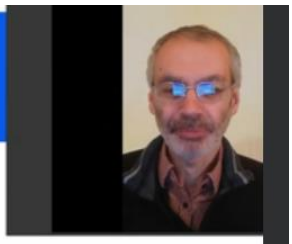
Exception Thrown in Critical Section

- When the exception is thrown:
  - The destructors are called for all objects in scope
  - The program flow jumps into the catch handler
  - The unlock call is never executed
  - The mutex remains locked
- All other threads which are waiting to lock the mutex are blocked
- If main() is joined on these blocked threads
  - main() will be blocked as well
  - The entire program is blocked

Drawbacks of std::mutex

- Calling lock() requires a corresponding call to unlock()
  - If not, the mutex will remain locked after the thread exits
- unlock() must always be called, even if
  - There are multiple paths through the critical section
  - An exception is thrown
- Relies on the programmer to get it right
- For these reasons, we do not normally use std::mutex directly

Mutex Wrapper Classes



- The C++ Library provides mutex wrapper classes
  - Classes with a mutex object as a private member
  - Defined in <mutex>
- These use the RAII idiom for managing resources
  - In this case, the resource is a lock on a mutex
  - The constructor locks the mutex
  - The destructor unlocks the mutex
- We create the wrapper class on the stack
  - The mutex will always be unlocked when the object goes out of scope
  - Including when an exception is thrown

std::lock_guard

- std::lock_guard is a very basic wrapper
  - Has a constructor and destructor only
- The constructor takes a mutex object as argument
  - Initializes its member from the argument
  - Locks it
- The destructor unlocks the mutex member

std::lock_guard

- std::lock_guard is a template class
- The template parameter is the type of the mutex

```
// Create a wrapper object for task_mutex
// which has type std::mutex
std::lock_guard<std::mutex> lck_guard(task_mutex);
```

std::lock_guard

- std::lock_guard is a template class
- The template parameter is the type of the mutex

```
// Create a wrapper object for task_mutex
// which has type std::mutex
```

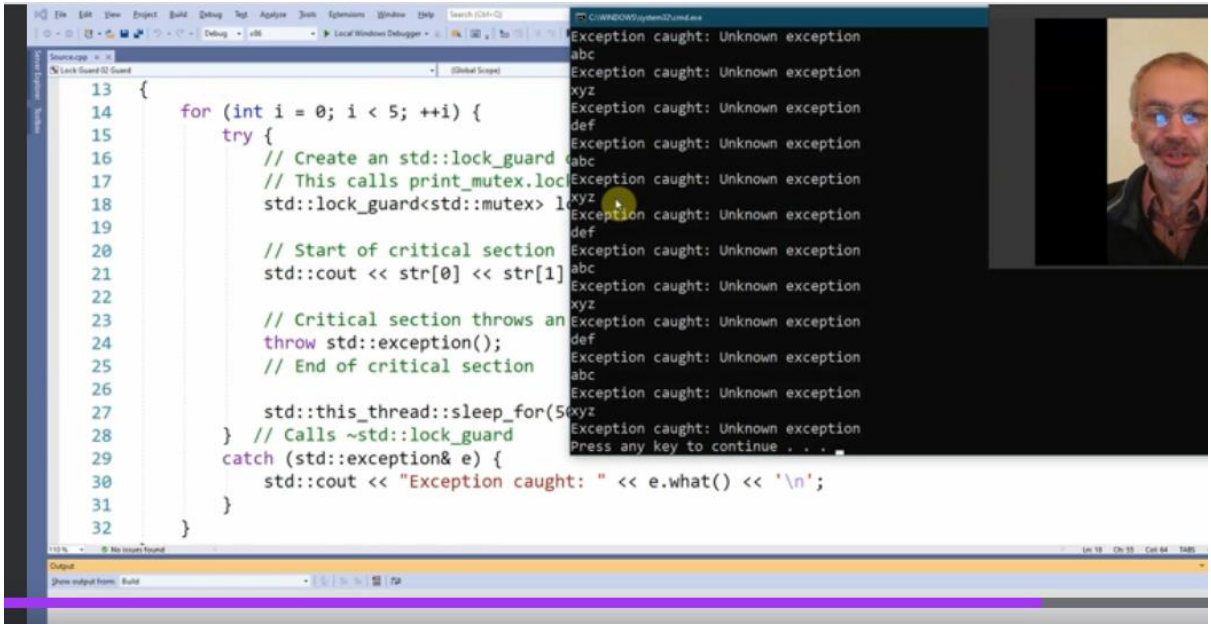
- In C++17, the compiler can deduce the mutex's type

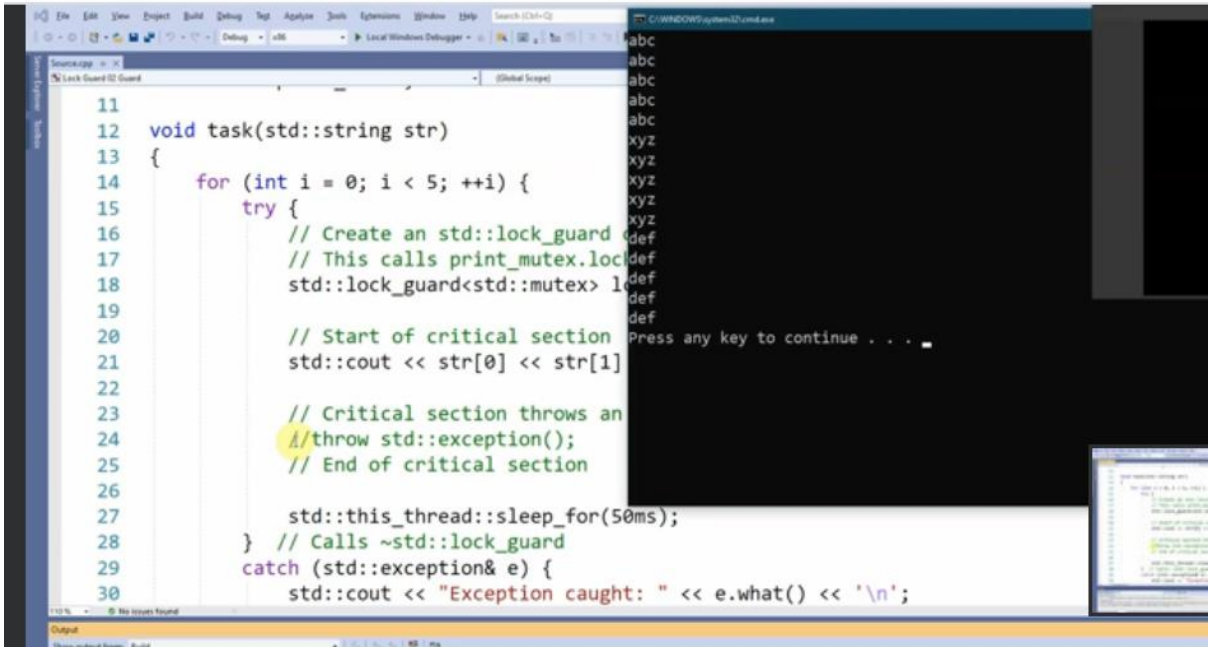
```
std::lock_guard lck_guard(task_mutex);
```

Output Example using std::lock_guard

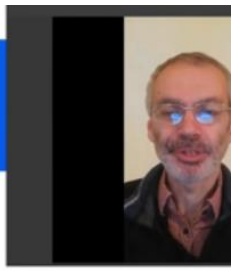
- Do not explicitly lock the mutex
- Create an std::lock_guard object
- Pass the mutex to its constructor

```
for (int i = 0; i < 5; ++i) {
    // Create an std::lock_guard object
    // This calls task_mutex.lock()
    std::lock_guard<std::mutex> lck_guard(task_mutex);

    // Critical section
    std::cout << str[0] << str[1] << str[2] << std::endl;
    // End of critical section

} // Calls ~std::lock_guard
```

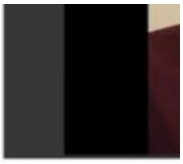




Output Example using std::lock_guard



- lck_guard is created
  - Its constructor calls lock()
- lck_guard goes out of scope
  - Its destructor calls unlock()
- If an exception is thrown
  - lck_guard's destructor is called and unlocks the mutex
- The mutex is never left unlocked
  - However, the mutex is still locked after the end of the critical section
  - Other threads cannot lock the mutex until lck_guard is destroyed



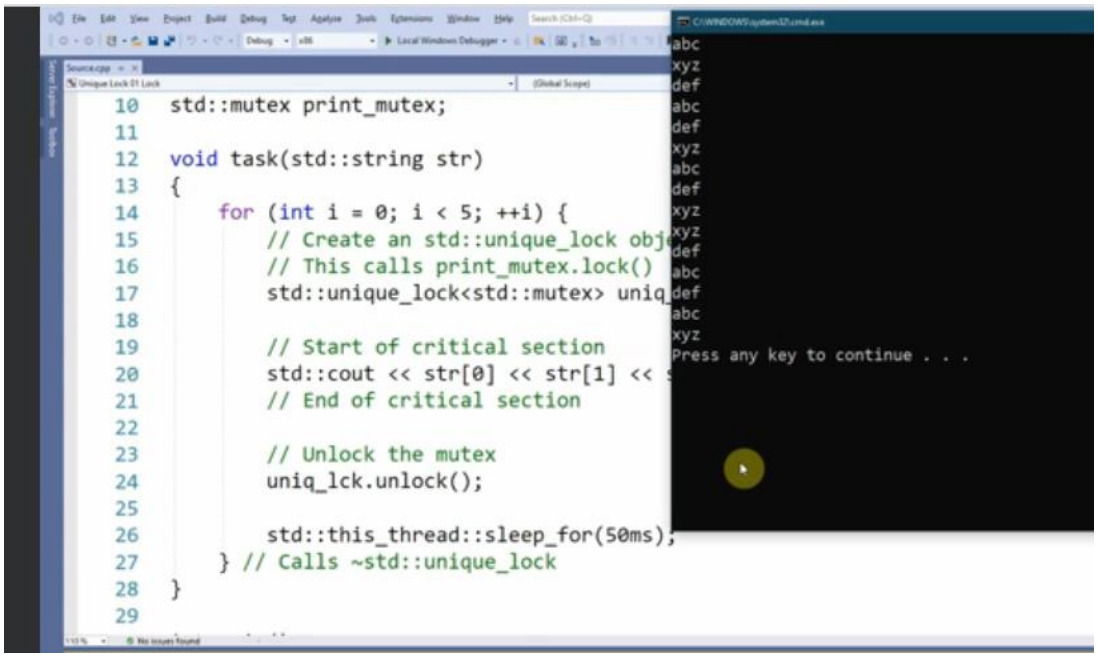
std::unique_lock

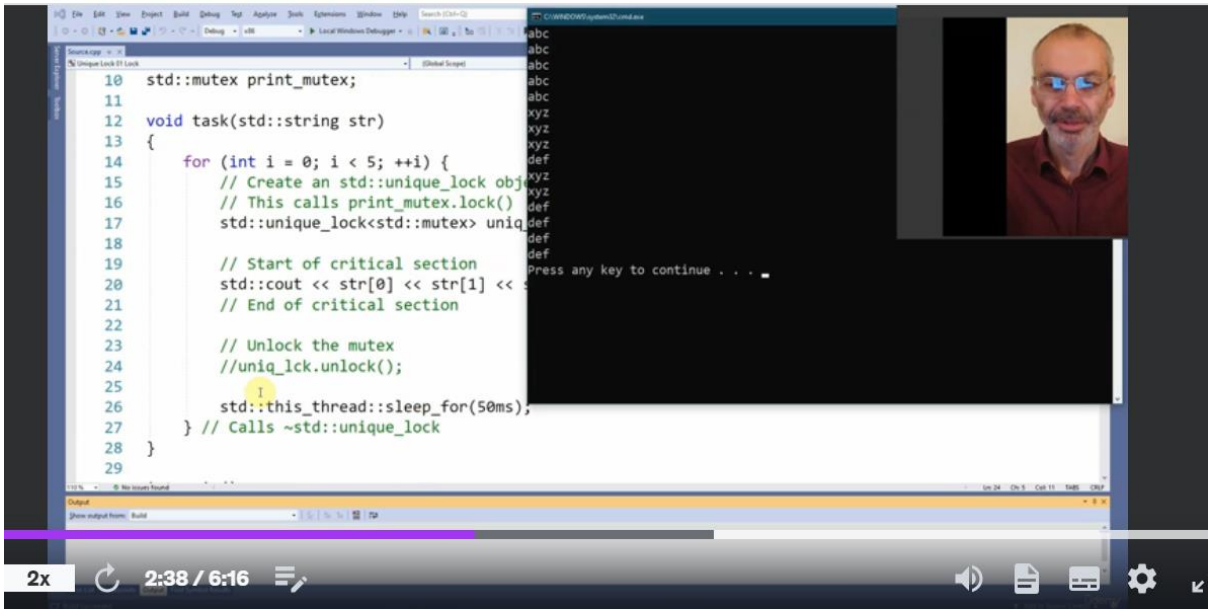
Output Example using std::lock_guard

```
for (int i = 0; i < 5; ++i) {
    // Create an std::lock_guard object
    // This calls task_mutex.lock()
    std::lock_guard<std::mutex> lck_guard(task_mutex);

    // Critical section
    std::cout << str[0] << str[1] << str[2] << std::endl;
    // End of critical section

    std::this_thread::sleep_for(50ms);
} // Calls ~std::lock_guard
```

std::unique_lock

- The same basic features as std::lock_guard
  - Mutex data member
  - Constructor locks the mutex
  - Destructor unlocks it
- It also has an unlock() member function
  - We can call this after the critical section
  - Avoids blocking other threads while we execute non-critical code
- If we do not call unlock(), the destructor will unlock the mutex
  - The lock is always released


std::mutex print_mutex;
void task(std::string str)
{
for (int i = 0; i < 5; ++i) {
// Create an std::unique_lock obj
// This calls print_mutex.lock()
std::unique_lock<std::mutex> uniq
// Start of critical section
std::cout << str[0] << str[1] <<
// End of critical section
// Unlock the mutex
uniq_lck.unlock();
std::this_thread::sleep_for(50ms);
} // Calls ~std::unique_lock
}
abc
xyz
def
Press any key to continue . . .


std::mutex print_mutex;
void task(std::string str)
{
for (int i = 0; i < 5; ++i) {
// Create an std::unique_lock obj
// This calls print_mutex.lock()
std::unique_lock<std::mutex> uniq
// Start of critical section
std::cout << str[0] << str[1] <<
// End of critical section
// Unlock the mutex
//uniq_lck.unlock();
std::this_thread::sleep_for(50ms);
} // Calls ~std::unique_lock
}
Press any key to continue . . .
2x
2:38 / 6:16

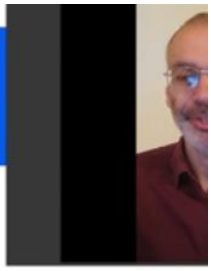
std::unique_lock Constructor Options

- The default
  - Call the mutex's lock() member function
- Try to get the lock
  - Do not wait if unsuccessful
  - (Timed mutex) Wait with a time-out if unsuccessful
- Do not lock the mutex
  - It will be locked later
  - Or the mutex is already locked

std::unique_lock Constructor Optional Second Argument

- std::try_lock
  - Calls the mutex's try_lock() member function
  - The owns_lock() member function checks if the mutex is locked
- std::defer_lock
  - Does not lock the mutex
  - Can lock it later by calling the lock() member function
  - Or by passing the std::unique_lock object to std::lock()
- std::adopt_lock
  - Takes a mutex that is already locked
  - Avoids locking the mutex twice

std::unique_lock and Move Semantics



- A std::unique_lock object cannot be copied
- It can be moved
  - The lock is transferred to another std::unique_lock object
  - Can only be done within the same thread
- We can write a function that creates a lock object and returns it
  - The function could lock different types of mutex, depending on its arguments
  - Factory design pattern

std::unique_lock vs std::lock_guard

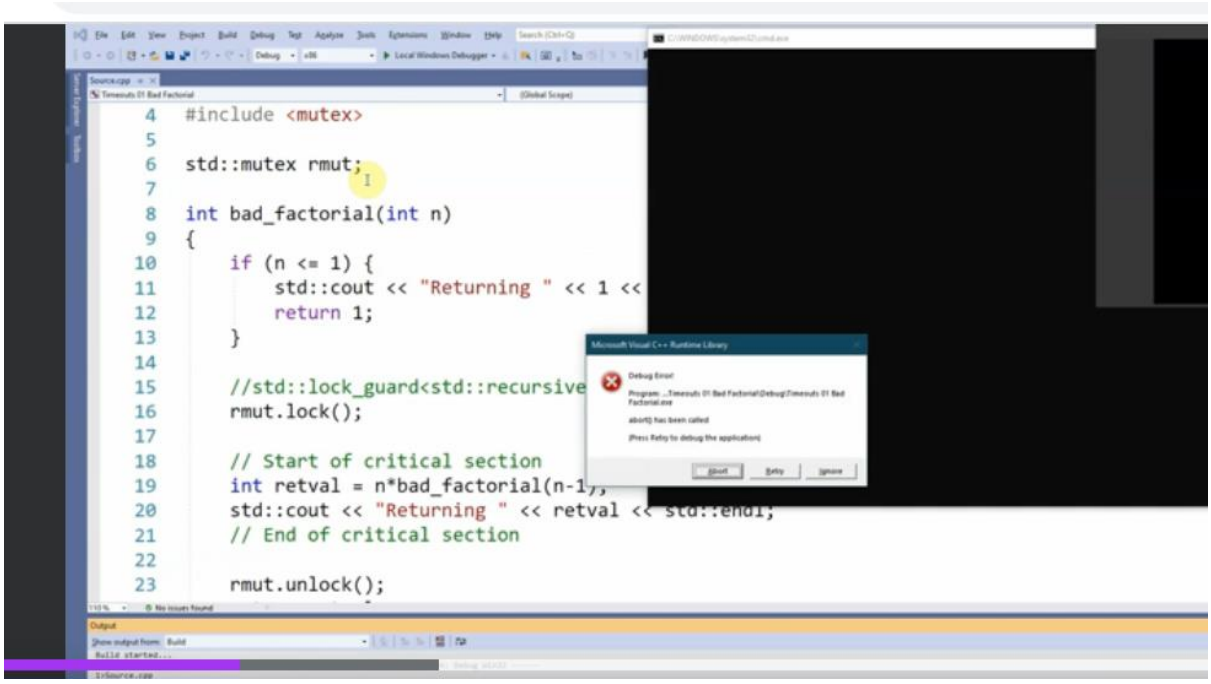
- std::unique_lock is much more flexible, but
  - Requires slightly more storage
  - Is slightly slower
- Recommendations:
  - Use lock_guard to lock a mutex for an entire scope
  - Use unique_lock if you need to unlock within the scope
  - Use unique_lock if you need the extra features

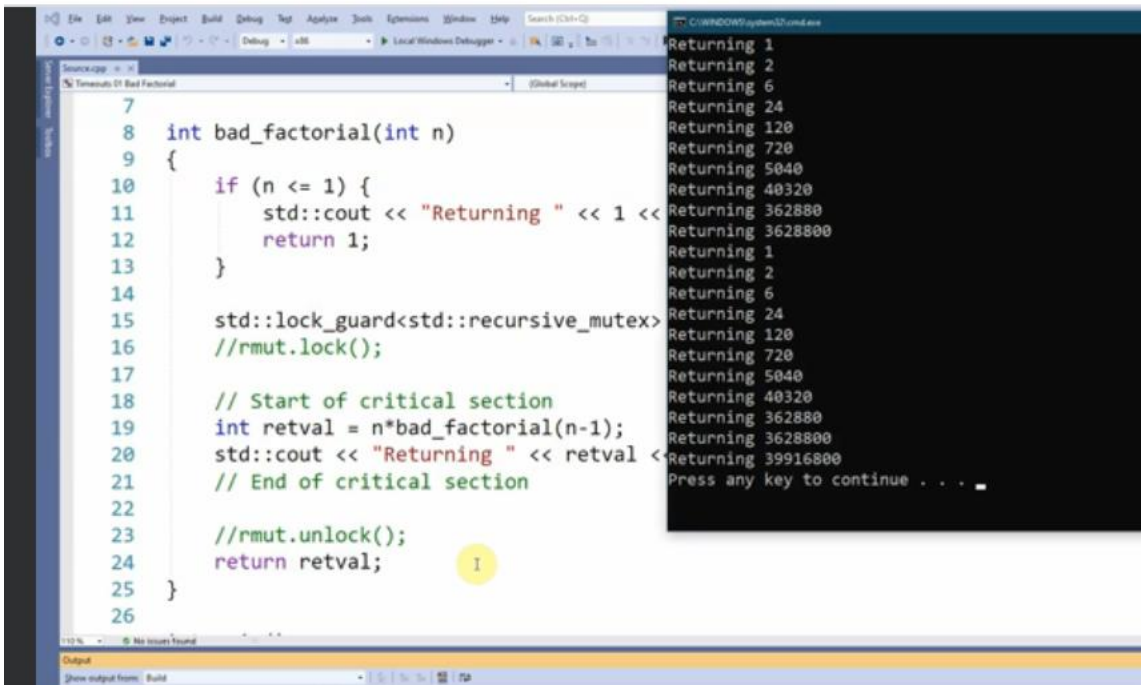
Multiple Locking

- A thread locks an std::mutex
- It must not call lock() again until it has called unlock()
  - Undefined behaviour
  - Usually the program blocks indefinitely

std::recursive_mutex

- Its lock() member function can be called repeatedly
  - Without calling unlock() first
  - For each lock() call, there must eventually be an unlock() call
- Normally a sign of bad design!





```
int bad_factorial(int n)
{
    if (n <= 1) {
        std::cout << "Returning " << 1 <<
        return 1;
    }

    std::lock_guard<std::recursive_mutex>
    //rmut.lock();

    // Start of critical section
    int retval = n*bad_factorial(n-1);
    std::cout << "Returning " << retval <<
    // End of critical section

    //rmut.unlock();
    return retval;
}
```

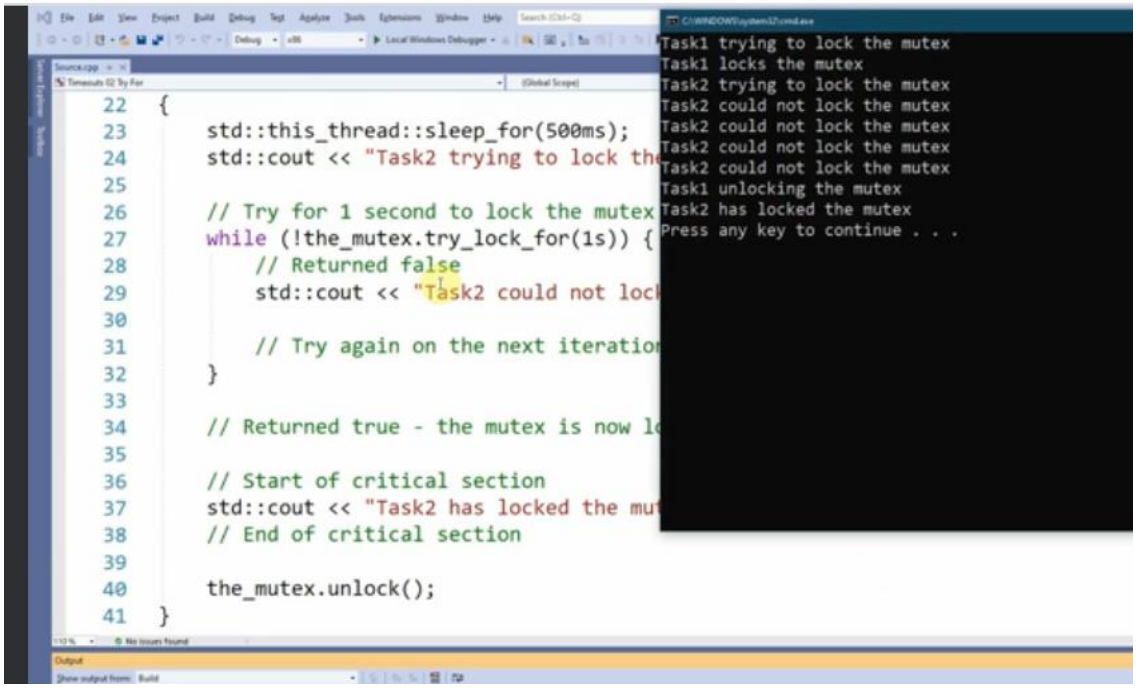
```
Returning 1
Returning 2
Returning 6
Returning 24
Returning 120
Returning 720
Returning 5040
Returning 40320
Returning 362880
Returning 3628800
Returning 1
Returning 2
Returning 6
Returning 24
Returning 120
Returning 720
Returning 5040
Returning 40320
Returning 362880
Returning 3628800
Returning 39916800
Press any key to continue . . .
```

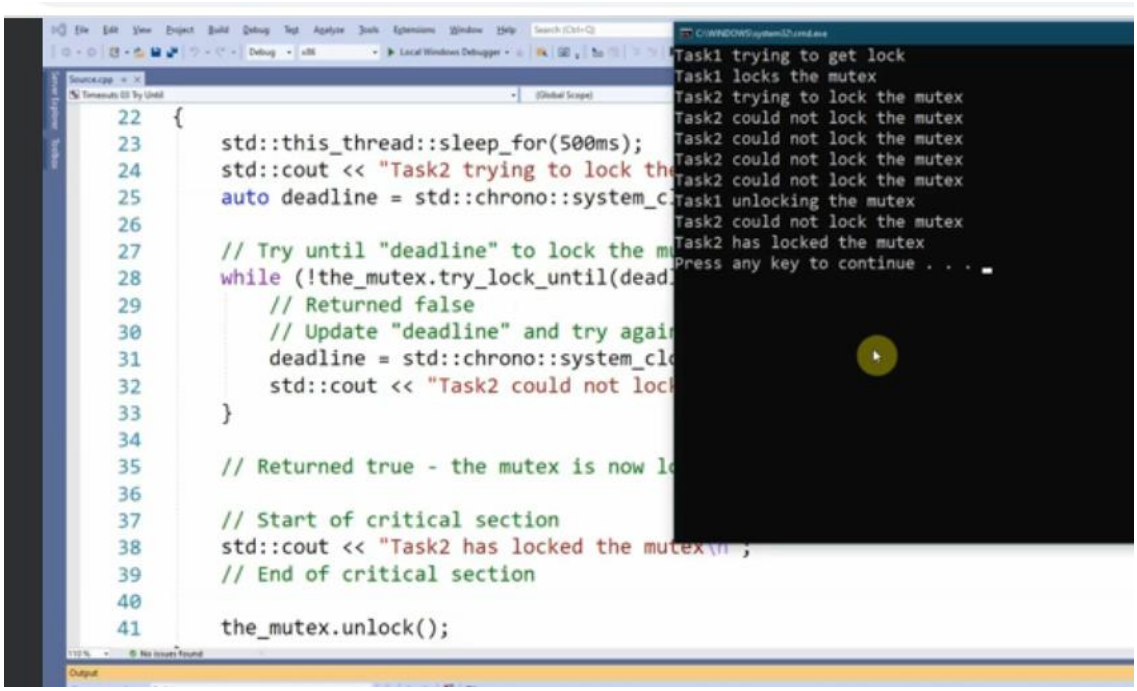
std::timed_mutex

- Similar to std::mutex, but with extra member functions
  - try_lock_for()
    - Keep trying to lock the mutex for a specified duration
  - try_lock_until()
    - Keep trying to lock the mutex until a specified time
- These return bool
  - True if the mutex was locked
  - Otherwise false

Other Mutexes with Timeouts

- There is an std::recursive_timed_mutex
- Has the same member functions
  - lock()
  - try_lock()
  - try_lock_for()
  - try_lock_until()





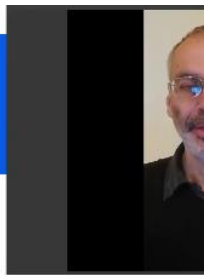
std::unique_lock

- std::unique_lock has member functions
  - try_lock_for()
  - try_lock_until()
- These are forwarded to the wrapped mutex
  - Will only compile if the mutex supports the operation

std::chrono Clocks

- chrono::system_clock
  - Gets time from operating system
  - May change erratically
  - Use it for time points
- chrono::steady_clock
  - Always increases steadily
  - Use it for measuring intervals
- try_lock_for() and try_lock_until() may return later than requested
  - Due to scheduling issues

Multiple Reader, Single Writer



- Financial data feed for infrequently traded stocks
  - Constantly accessed to get the latest price
  - The price rarely changes
- Audio/video buffers in multimedia players
  - Constantly accessed to get the next frame
  - Occasionally updated with a block of data
- Shared data
  - Must protect against a data race

Data Race

- A "data race" occurs when:
  - Two or more threads access the same memory location
  - And at least one of the threads modifies it
  - Potentially conflicting accesses to the same memory location
- Only safe if the threads are synchronized
  - One thread accesses the memory location at a time
  - The other threads have to wait until it is safe for them to access it
  - In effect, the threads execute sequentially while they access it
- A data race causes undefined behaviour
  - The program is not guaranteed to behave consistently

Multiple Reader, Single Writer

- Concurrent accesses:
- High probability of a reader and another reader
  - No locking required
- Low probability of a writer and a reader
  - Locking required
- Low probability of a writer and another writer
  - Locking required

Multiple Reader, Single Writer

- With std::mutex, all threads are synchronized
- They must execute their critical sections sequentially
  - Even when it is not necessary
- Loss of concurrency reduces performance

Read-write Lock

- It would be useful to have "selective" locking
  - Lock when there is a thread which is writing
  - Do not lock when there are only reading threads
  - Often called a "read-write lock"

Shared Mutex

std::shared_mutex

- std::shared_mutex is defined in <shared_mutex>
- It can be locked in two different ways:
- Exclusive lock
  - No other thread may acquire a lock
  - No other thread can enter a critical section
- Shared lock
  - Other threads may acquire a shared lock
  - They can execute critical sections concurrently

Shared lock

- std::shared_lock<std::shared_mutex>
- A thread which has a shared lock can enter a critical sectio
- It can only acquire a shared lock if there are no exclusive locks
  - If another thread has an exclusive lock
  - This thread must wait until the exclusive lock is released

shared_mutex usage

```
std::shared_mutex shmut;

void write()
{
   std::lock_guard lck_guard(shmut);      // Write thread with exclusive lock
   ...
}

void read()
{
   std::shared_lock sh_lck(shmut);        // Read thread with shared lock
   ...
}
```

std::shared_mutex Member Functions

- Exclusive locking
  - lock()
  - try_lock()
  - unlock()
- Shared locking
  - lock_shared()
  - try_lock_shared()
  - unlock_shared()

Data Race Avoidance

- The writer thread cannot get an exclusive lock
  - Until all other threads release their locks
  - Those threads have now left their critical sections
- The writer thread acquires an exclusive lock
  - It enters the critical section
  - Reader threads cannot get a shared lock
  - Writer threads cannot get an exclusive lock
  - Until this thread releases its lock
- The writer thread releases its exclusive lock
  - It has now left its critical section

Data Race Avoidance

- The reader thread cannot get a shared lock
  - Until a writer thread releases its exclusive lock
  - The writer thread has now left its critical section
- The reader thread acquires a shared lock
  - It enters the critical section
  - Other reader threads can also get a shared lock
- There is no scenario in which there is a data race
  - Reader and writer threads cannot interleave in a critical section

Pros and Cons of std::shared_mute

- Uses more memory than std::mutex
- Slower than std::mutex
- Best suited to situations where
  - Reader threads greatly outnumber writer threads
  - Read operations take a long time

Shared Data

- Global variable
  - Accessible to all code in the program
- Static variable at namespace scope
  - Accessible to all code in the translation unit
- Class member which is declared static
  - Potentially accessible to code which calls its member functions
  - (If public, accessible to all code)
- Local variable which is declared static
  - Accessible to all code which calls that function

Shared Data Initialization

- Global variable
- Static variable at namespace scope
- Static data member of class
  - All are initialized when the program starts
  - At that point, only one thread is running
  - There cannot be a data race

Static Local Variable

- Initialized after the program starts
- When the declaration is reached

```
void func() {
  ...
  // Static local variable
  static std::string str("xyz");
  ...
}
```

- Two or more threads may call the constructor concurrently
- Is there a data race?

Data Race

- A "data race" occurs when:
  - Two or more threads access the same memory location
  - And at least one of the threads modifies it
  - Potentially conflicting accesses to the same memory location
- Only safe if the threads are synchronized
  - One thread accesses the memory location at a time
  - The other threads have to wait until it is safe for them to access it
  - In effect, the threads execute sequentially while they access it
- A data race causes undefined behaviour
  - The program is not guaranteed to behave consistently

Static Local Variable Initialization Before C++11

- No language support
  - The behaviour was undefined
- Lock a mutex?
  - Required on every pass through the declaration
  - Very inefficient

Static Local Variable Initialization in C++11

- The behaviour is now well-defined
- Only one thread can initialize the variable
  - Any other thread that reaches the declaration is blocked
  - Must wait until the first thread has finished initializing the variabl
  - The threads are synchronized by the implementation
  - No data race
- Subsequent modifications
  - The usual rules for shared data
  - There will be a data race, unless we protect against one

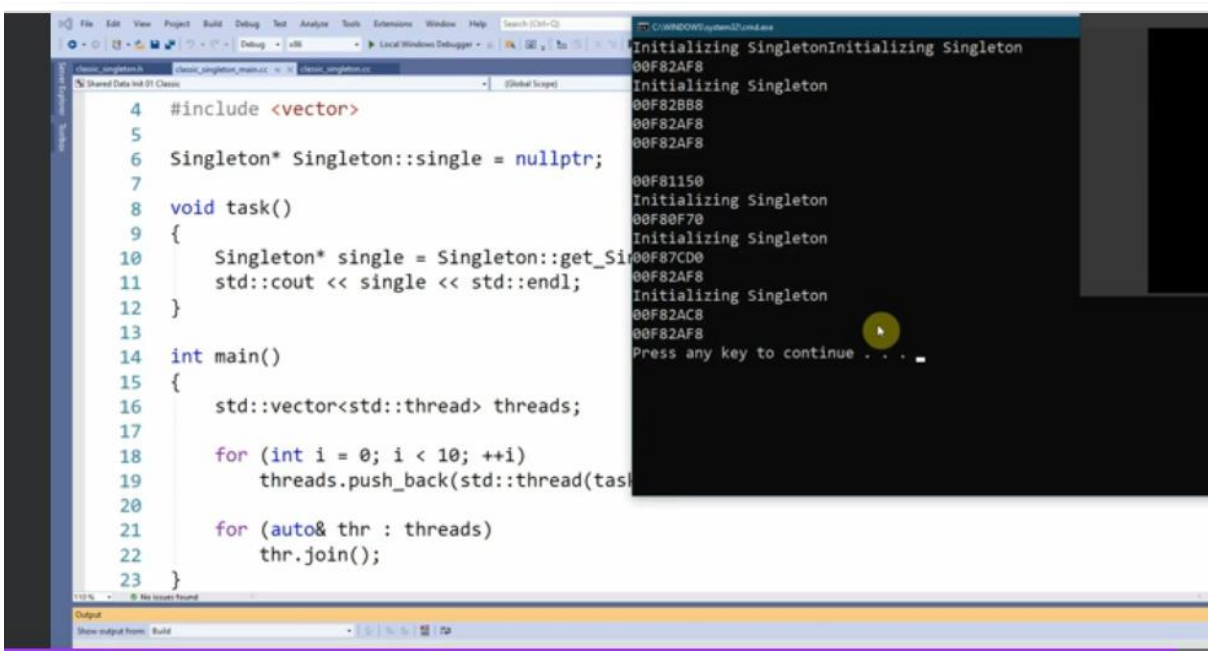
Singleton Class

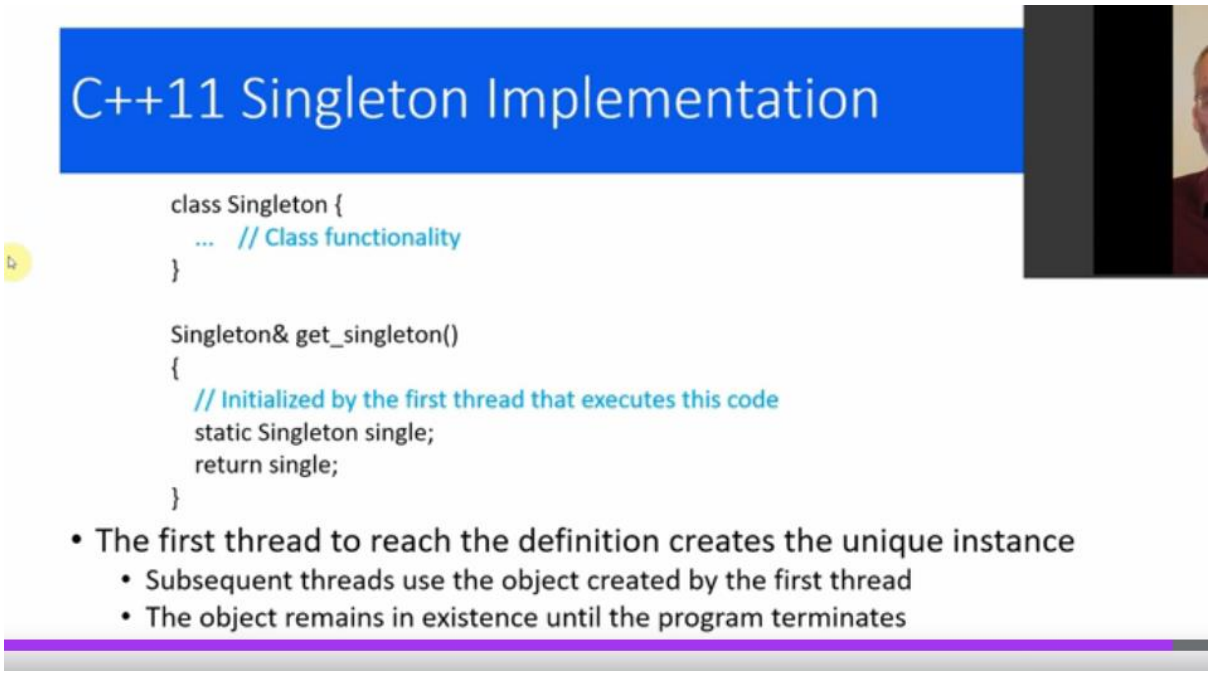
- Used to implement the Singleton design pattern
- A Singleton class has only a single instance
  - e.g. a logger class that maintains an audit trail
- Its constructor is private
- The copy and move operators are deleted
  - The program cannot create more objects
- A static member function returns the unique instance
  - If the instance does not already exist, it is created and initialized
  - Otherwise, the existing object is returned

Classic Singleton Implementation

```
class Singleton {
   // Pointer to the unique instance
   static Singleton *single;

public:
    // Static member function which returns the unique instance
   static Singleton* get_singleton() {
      if (single == nullptr)
         single = new Singleton;
      return single;
   }

   ... // Class functionality
}
```

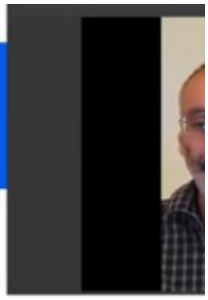
- This has a data race


#include <vector>
Singleton* Singleton::single = nullptr;
void task()
{
Singleton* single = Singleton::get_Si
std::cout << single << std::endl;
}
int main()
{
std::vector<std::thread> threads;
for (int i = 0; i < 10; ++i)
threads.push_back(std::thread(tas
for (auto& thr : threads)
thr.join();
}
Initializing SingletonInitializing Singleton
00F82AF8
Initializing Singleton
00F82BB8
00F82AF8
00F82AF8
00F81150
Initializing Singleton
00F80F70
Initializing Singleton
00F87CD0
00F82AF8
Initializing Singleton
00F82AC8
00F82AF8
Press any key to continue . . .


C++11 Singleton Implementation
class Singleton {
... // Class functionality
}
Singleton& get_singleton()
{
// Initialized by the first thread that executes this code
static Singleton single;
return single;
}
• The first thread to reach the definition creates the unique instance
• Subsequent threads use the object created by the first thread
• The object remains in existence until the program terminates


Thread-local Variables

Thread-local Variables



- C++ supports thread-local variables
  - Same as static and global variables
  - However, there is a separate object for each thread
  - With a static variable, there is a single object which is shared by all threads
- We use the thread_local keyword to declare them

Thread-local Variables

- Global variables or at namespace scope
- Data members of a class
- Local variables in a function

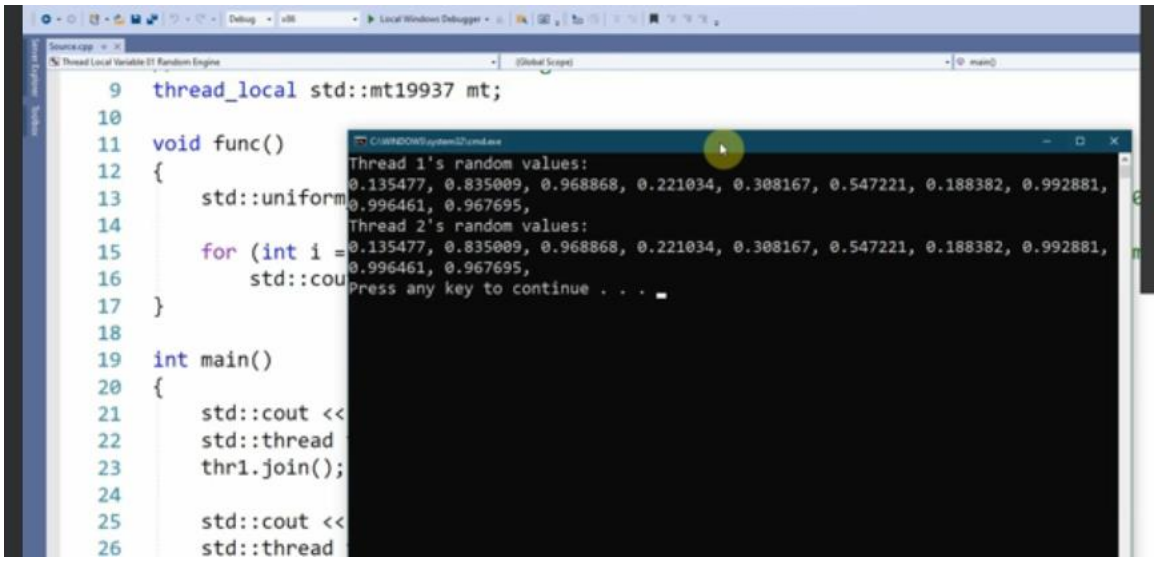
Thread-local Variable Lifetimes

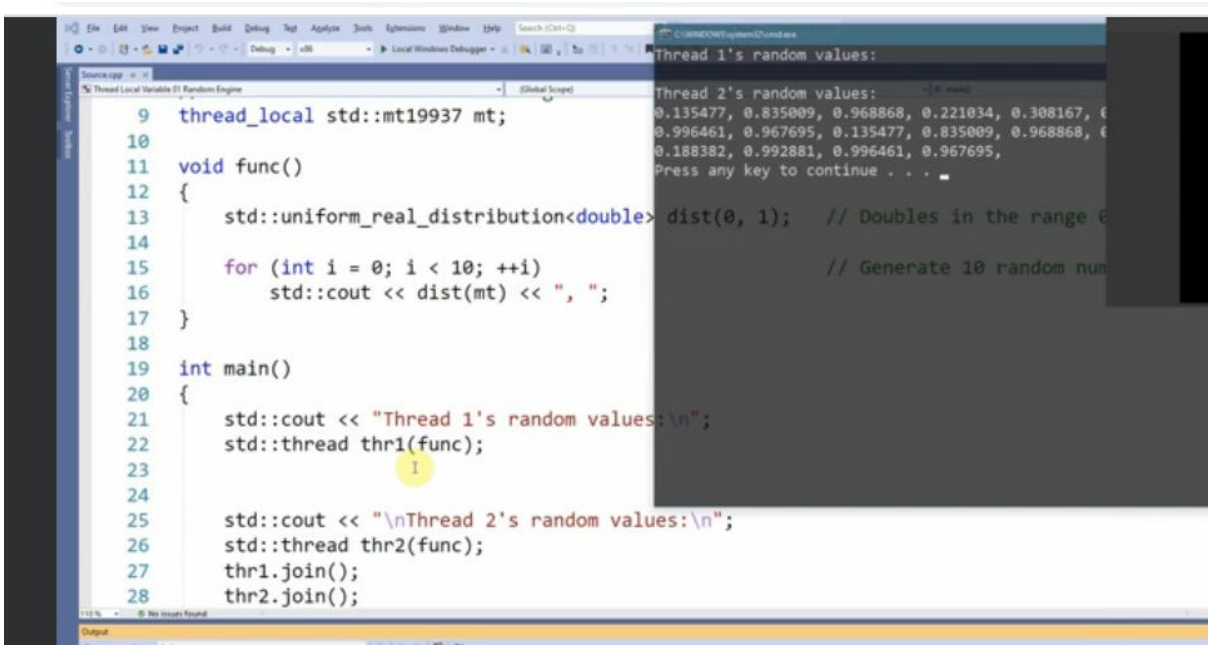
- Global and namespace scope
  - Always constructed at or before the first use in a translation unit
  - It is safe to use them in dynamic libraries (DLLs)
- Local variables
  - Initialized in the same way as static local variables
- In all cases
  - Destroyed when the thread completes its execution

Thread-local Variable Example

- We can make a random number engine thread-local
  - This gives each thread its own object
- This ensures that each thread generates the same sequence
  - Useful for testing

```
// Thread-local random number engine
std::thread_local mt19937 mt;

void func()
{
  std::uniform_real_distribution<double> dist(0, 1);          // Doubles in the range 0 to 1

  for (int i = 0; i < 10; ++i)                                // Generate 10 random numbers
    std::cout << dist(mt) << ", ";
}
```





Lazy Initialization

- Common pattern in functional programming
- A variable is only initialized when it is first used
- This is useful when the variable is expensive to construct
  - e.g. it sets up a network connection
- Can be used in multi-threaded code
  - But we need to avoid data races

```cpp
// Lazy initialization (single-threaded)
class Test {
    // ...
public:
    void func() { /*...*/ }
};

Test *ptest = nullptr;          // Variable to be lazily initialized

void process() {
    if (!ptest)                 // First time variable has been used
        ptest = new Test;       // Initialize it
    ptest->func();              // Use it
}
```

```cpp
// Lazy initialization (multi-threaded)
// Inefficient (always locks the mutex)
#include <mutex>

class Test;

Test *ptest = nullptr;          // Variable to be lazily initialized
std::mutex mut;

void process() {
    std::unique_lock<std::mutex> uniq_lck(mut);       // Protect ptest

    if (!ptest)
        ptest = new Test;
    uniq_lck.unlock();
    ptest->func();
}
```

Thread-safe Lazy Initialization

- Every thread that calls process() locks the mutex
  - Locking the mutex blocks every other thread that calls process()
- The lock is only needed while ptest is being initialized
  - Once ptest has been initialized, locking the mutex is unnecessary
  - Causes a loss of performance

Double-checked Locking Algorithm

- More efficient version of thread-safe lazy initialization
- If ptest is not initialized
  - Lock the mutex
  - If ptest is not initialized, initialize it
  - Unlock the mutex
  - Use ptest
- Otherwise
  - Use ptest
- ptest is checked twice (why?)

Double-checked Locking

```
void process() {
  if (!ptest) {                          // First check of ptest
    std::lock_guard lck_guard(mut);

    if (!ptest)                          // Second check of ptest
      ptest = new Test;                  // Initialize ptest
  }

  // Use ptest
  // ...
}
```

Why Two Checks?

```
if (!ptest) {                          // First check of ptest (1
   std::lock_guard lck_guard(mut);     // Lock the mutex (2)

   ptest = new Test;                   // Initialize ptest
}
// Use ptest ...
```

- Statement (1) checks ptest
- Statement (2) locks the mutex
- Another thread could interleave between these operations
  - Race condition

Why Two Checks?

```
if (!ptest) {
   std::lock_guard lck_guard(mut);     // First check of ptest (1)
   ptest = new Test;                   // Lock the mutex (2)
}                                      // Initialize ptest
// Use ptest ...
```

- Thread A checks ptest, which is null
- Thread B checks ptest, which is null
- Thread B locks the mutex
- Thread B initializes ptest
- Thread B unlocks the mutex
- Thread A locks the mutex
- Thread A initializes ptest

Double-checked Locking

```
void process() {
  if (!ptest) {                          // First check of ptest
    std::lock_guard lck_guard(mut);

    if (!ptest)                          // Second check of ptest
      ptest = new Test;                  // Initialize ptest
  }

  // Use ptest
  // ...
}
```

Is That Not Enough?

- There is still a race condition

  ptest = new Test;

- The initialization of ptest involves several operations
  - Allocate enough memory to store a Test object
  - Construct a Test object in the memory
  - Store the address in ptest
- C++ allows these to be performed in a different order, e.g.
  - Allocate enough memory to store a Test object
  - Store the address in ptest
  - Construct a Test object in the memory

Undefined behaviour

- Thread A checks ptest and locks the mutex
- Thread A allocates the memory and assigns to ptest

```
ptest = new sizeof(Test);
```

- However, it has not yet called the constructor
- Thread B checks ptest and it is not null
- Thread B does not lock the mutex
- Thread B jumps out of the if statement
- Thread B calls a member function of an uninitialized object
  - Undefined behaviour

std::call_once

- One way to solve this is to use std::call_once()
  - A given function is only called once
  - It is done in one thread
  - The thread cannot be interrupted until the function call completes
- We use it with a global instance of std::once_flag
- We pass the instance and the function to std::call_once()

Double-checked Locking in C++17

- C++17 defines the order of initialization
  - Allocate enough memory to store a Test object
  - Construct a Test object in the memory
  - Store the address in ptest

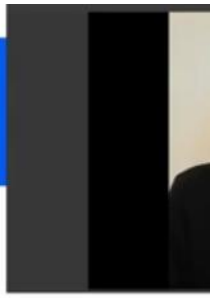
```
ptest = new Test;
```

- Double-checked locking no longer causes a data race

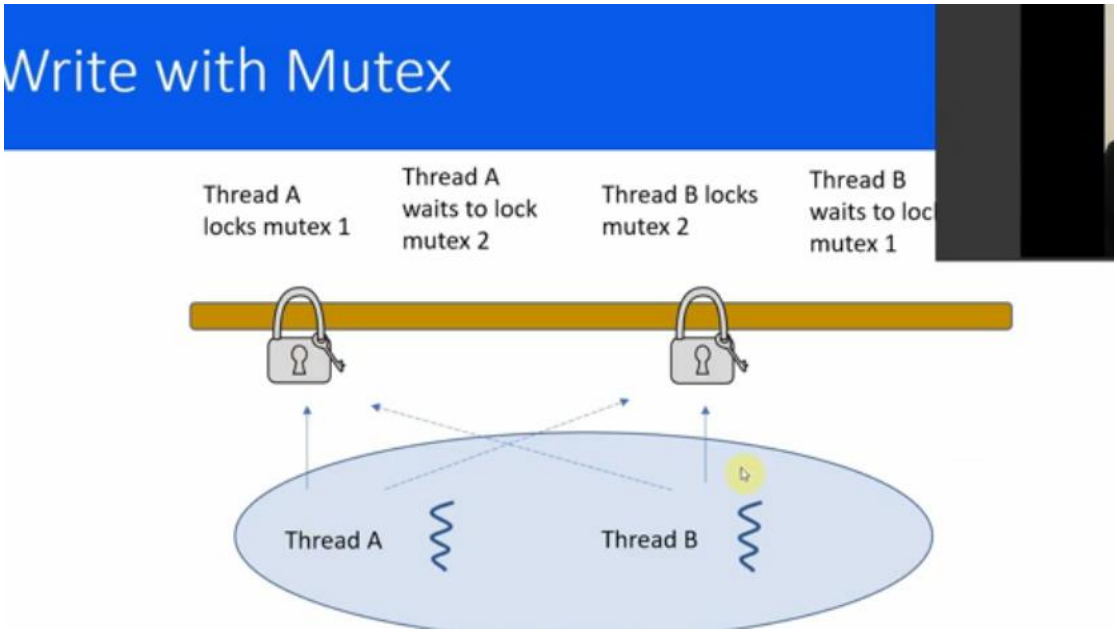
Conclusion

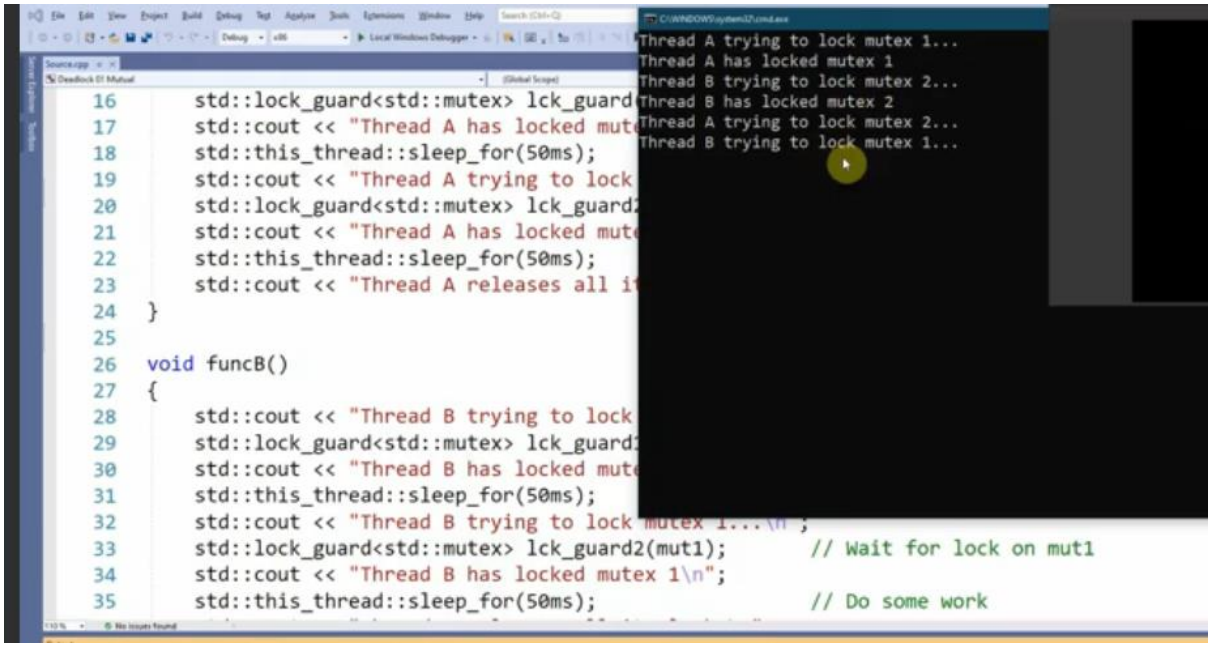
- Four ways to do thread-safe lazy initialization
  - Naive use of a mutex
  - Use std::call_once()
  - Double-checked locking with a C++17 compiler or later
  - Meyers singleton with static local variable
- Recommendations
  - Use Meyers singleton, if ptest is not required outside the function
  - Otherwise, use std::call_once()

Deadlock



- A thread is deadlocked when it cannot run
- Often used to refer to "mutual deadlock"
  - Two or more threads are waiting for each other
  - Thread A waits for thread B to do something
  - Thread B is waiting for A to do something
  - Threads A and B are waiting for an event that can never happen
- The classic example involves waiting for mutexes


Write with Mutex
Thread A
locks mutex 1
Thread A
waits to lock
mutex 2
Thread B locks
mutex 2
Thread B
waits to loc
mutex 1
Thread A
Thread B

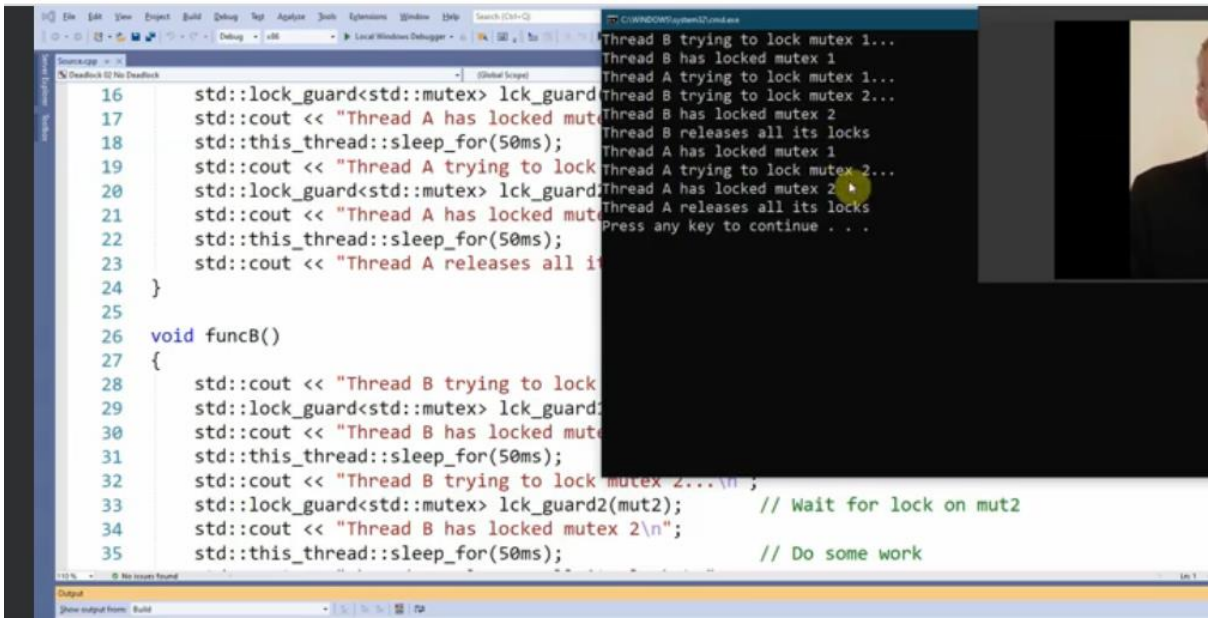

std::lock_guard<std::mutex> lck_guard
std::cout << "Thread A has locked mut
std::this_thread::sleep_for(50ms);
std::cout << "Thread A trying to lock
std::lock_guard<std::mutex> lck_guard2
std::cout << "Thread A has locked mut
std::this_thread::sleep_for(50ms);
std::cout << "Thread A releases all i
}
void funcB()
{
std::cout << "Thread B trying to lock
std::lock_guard<std::mutex> lck_guard1
std::cout << "Thread B has locked mut
std::this_thread::sleep_for(50ms);
std::cout << "Thread B trying to lock mutex 1...\n";
std::lock_guard<std::mutex> lck_guard2(mut1); // Wait for lock on mut1
std::cout << "Thread B has locked mutex 1\n";
std::this_thread::sleep_for(50ms); // Do some work
Thread A trying to lock mutex 1...
Thread A has locked mutex 1
Thread B trying to lock mutex 2...
Thread B has locked mutex 2
Thread A trying to lock mutex 2...
Thread B trying to lock mutex 1...

Mutual Deadlock

- Can also occur when waiting for
  - The result of a computation performed by another thread
  - A message sent by another thread
  - A GUI event produced by another thread
- The second most common problem in multi-threading code
- Often caused by threads trying to lock mutexes in different orders

Deadlock Avoidance

- A simple way to avoid deadlock
- Both threads try to acquire the locks in the same order
  - Thread A and thread B both try to lock mutex1 first
  - The successful thread then tries to lock mutex2



Deadlock Avoidance

- This is not ideal
  - Relies on the programmer
  - May not be feasible in large programs
- We will look at some better approaches in the next lecture

Deadlock Practical

Dining Philosophers

0:33 / 7:46

Dining Philosophers Rules

- A philosopher has two states: thinking and eating
- Each fork can only be held by one philosopher at a time
  - A philosopher can only pick up one fork at a time
  - A philosopher must pick up both forks before they can eat
  - When a philosopher finishes eating, they put down both forks immediately
  - A philosopher may pick up a fork as soon as it is put down by another
- A philosopher has no awareness of other philosophers' actions
- If a philosopher does not eat at all, they will die of starvation

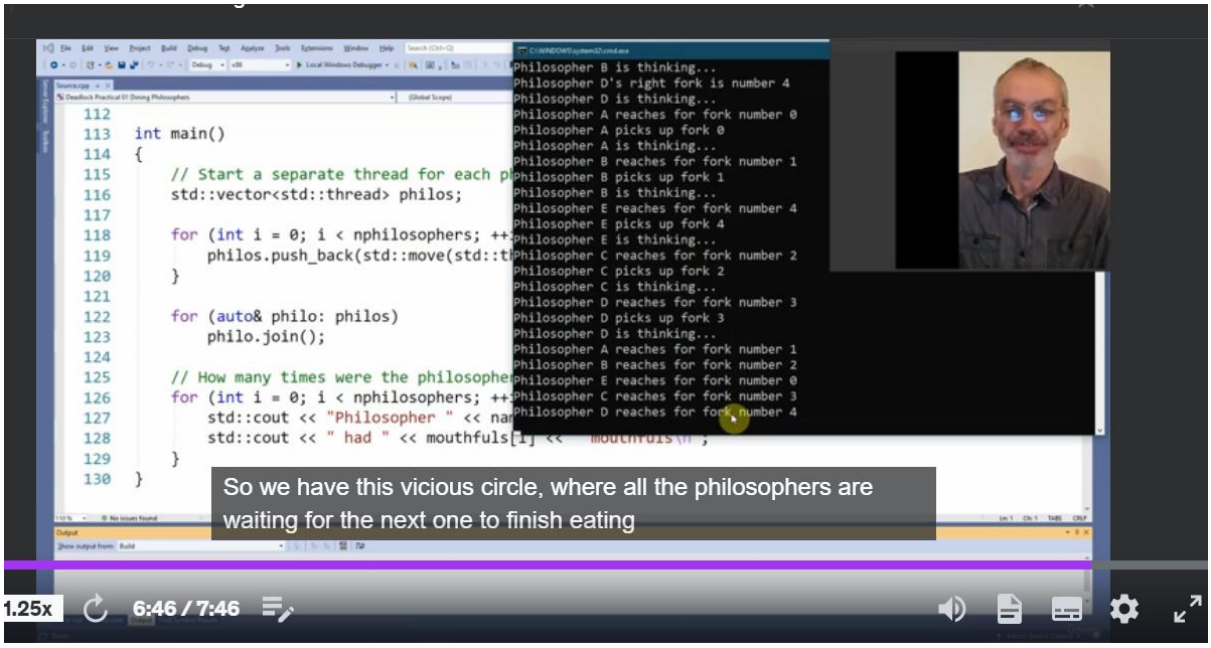
Dining Philosophers

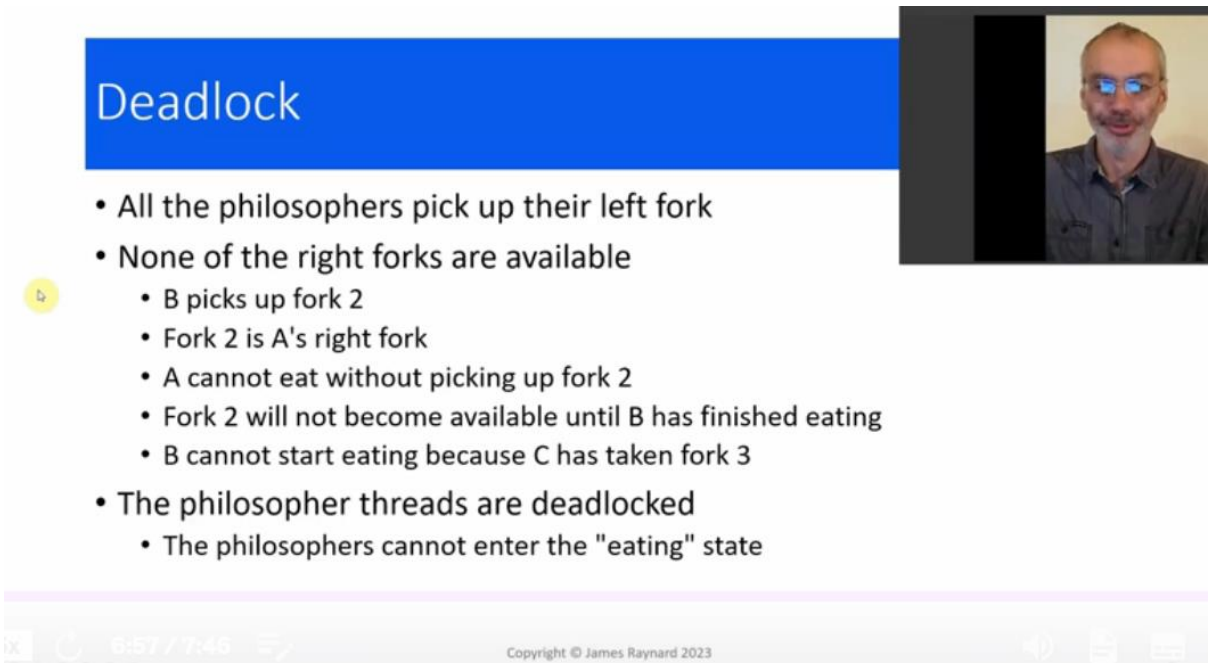
- Intended scenario
  - Philosopher thinks
  - Philosopher picks up left fork
  - Philosopher thinks
  - Philosopher picks up right fork
  - Philosopher eats
  - Philosopher puts down both forks
  - Philosopher thinks

Implementation

- A separate thread for each philosopher
- Each fork has an associated mutex

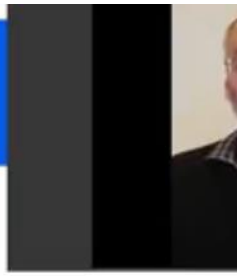
```
// A mutex prevents more than one philosopher picking up the same fork
std::mutex fork_mutex[nforks];

// A philosopher thread can only pick up a fork if it can lock the corresponding mutex
// Try to pick up the left fork
fork_mutex[lfork].lock();

// Try to pick up the right fork
fork_mutex[rfork].lock();

// Succeeded - this philosopher can now eat
// ...
```


int main()
{
// Start a separate thread for each
std::vector<std::thread> philos;
for (int i = 0; i < nphilosophers;
philos.push_back(std::move(std::
}
for (auto& philo: philos)
philo.join();
// How many times were the philosophe
for (int i = 0; i < nphilosophers;
std::cout << "Philosopher " << na
std::cout << " had " << mouthfuls[i] <<
}
}
Philosopher B is thinking...
Philosopher D's right fork is number 4
Philosopher D is thinking...
Philosopher A reaches for fork number 0
Philosopher A picks up fork 0
Philosopher A is thinking...
Philosopher B reaches for fork number 1
Philosopher B picks up fork 1
Philosopher B is thinking...
Philosopher E reaches for fork number 4
Philosopher E picks up fork 4
Philosopher E is thinking...
Philosopher C reaches for fork number 2
Philosopher C picks up fork 2
Philosopher C is thinking...
Philosopher D reaches for fork number 3
Philosopher D picks up fork 3
Philosopher D is thinking...
Philosopher A reaches for fork number 1
Philosopher B reaches for fork number 2
Philosopher E reaches for fork number 0
Philosopher C reaches for fork number 3
Philosopher D reaches for fork number 4
So we have this vicious circle, where all the philosophers are waiting for the next one to finish eating
1.25x
6:46 / 7:46


Deadlock
• All the philosophers pick up their left fork
• None of the right forks are available
• B picks up fork 2
• Fork 2 is A's right fork
• A cannot eat without picking up fork 2
• Fork 2 will not become available until B has finished eating
• B cannot start eating because C has taken fork 3
• The philosopher threads are deadlocked
• The philosophers cannot enter the "eating" state
Copyright © James Raynard 2023

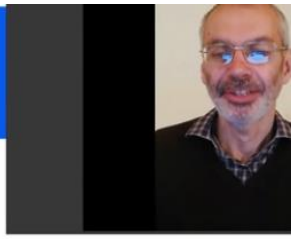

Deadlock Avoidance

Deadlock Avoidance



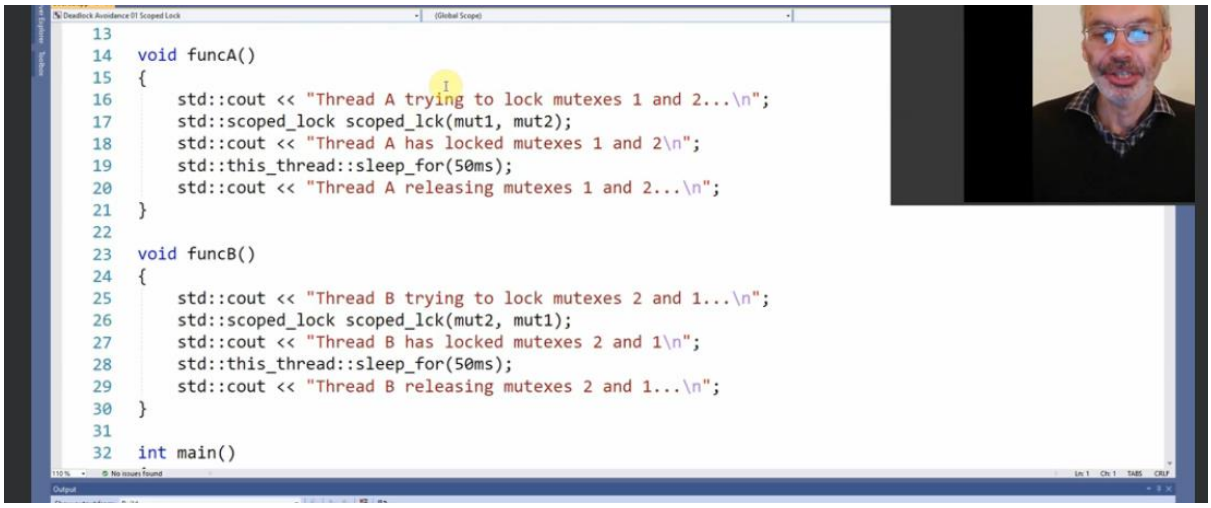
- Make the threads acquire locks in the same order
  - Relies on the programmer
- Lock multiple mutexes in a single operation
  - Thread A locks mut1 and mut2
  - Thread B cannot lock mut2 or mut1 during this operation
  - A much better solution
- C++ provides library features for this

std::scoped_lock

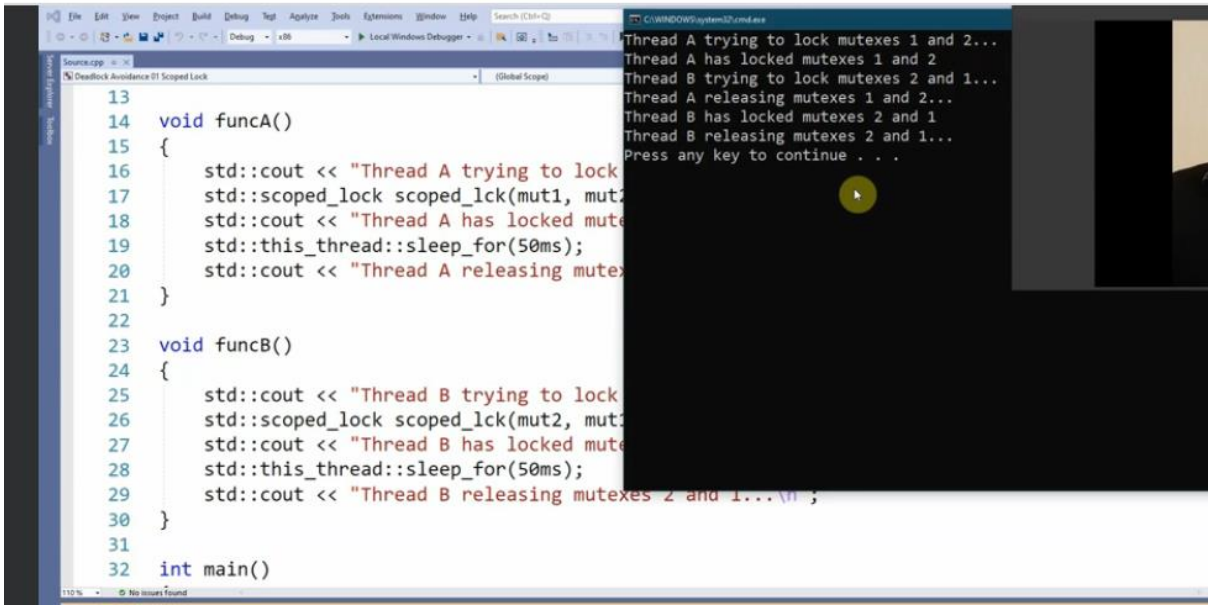


- C++17 has std::scoped_lock
- Very similar to std::lock_guard
  - Except it can lock more than one mutex at the same time

    std::scoped_lock scope_lck(mut1, mut2, ...);
- The mutexes are locked in the order given in the constructor call
  - In the destructor, the mutexes are unlocked in the reverse order
- This avoids the possibility of deadlock with multiple mutexes




```
13
14  void funcA()
15  {
16      std::cout << "Thread A trying to lock mutexes 1 and 2...\n";
17      std::scoped_lock scoped_lck(mut1, mut2);
18      std::cout << "Thread A has locked mutexes 1 and 2\n";
19      std::this_thread::sleep_for(50ms);
20      std::cout << "Thread A releasing mutexes 1 and 2...\n";
21  }
22
23  void funcB()
24  {
25      std::cout << "Thread B trying to lock mutexes 2 and 1...\n";
26      std::scoped_lock scoped_lck(mut2, mut1);
27      std::cout << "Thread B has locked mutexes 2 and 1\n";
28      std::this_thread::sleep_for(50ms);
29      std::cout << "Thread B releasing mutexes 2 and 1...\n";
30  }
31
32  int main()
```



std::scoped_lock Caveat

- scoped_lock can be used with a single mutex

```
std::scoped_lock scoped_lck(mut);
```

- It is easy to accidentally omit the argument

```
std::scoped_lock scoped_lck;
```

- This will compile and run, but not actually perform any locking
  - May cause an unexpected data race

Deadlock Avoidance Before C++17

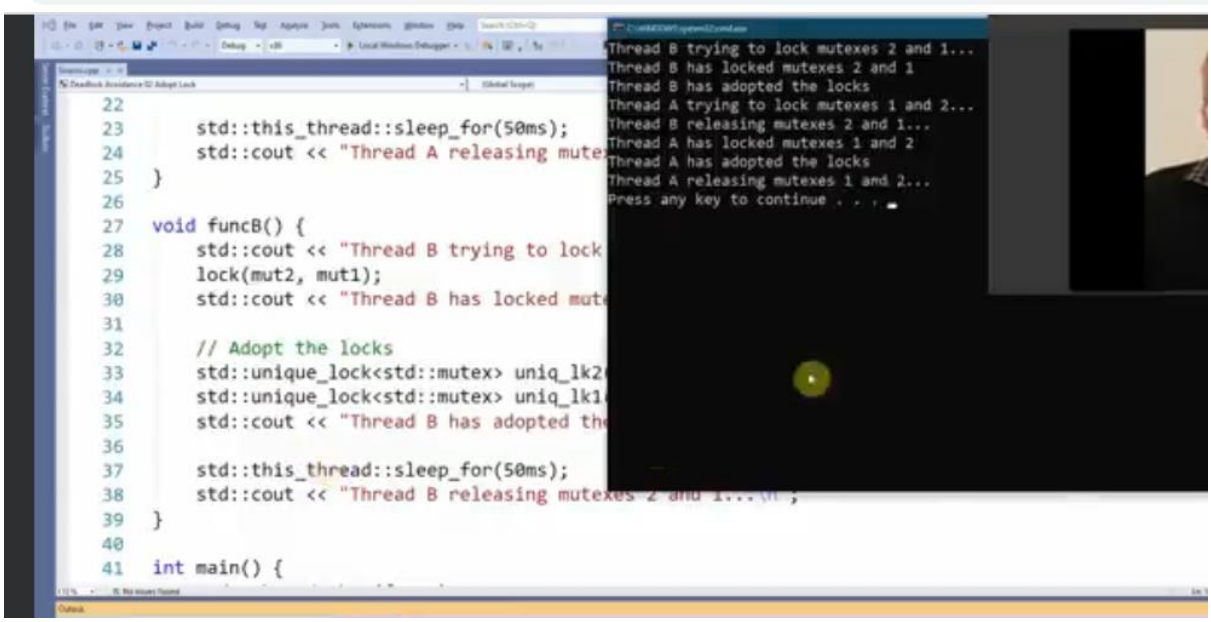
- Use the std::lock() function
  - It can lock multiple mutexes in a single operation

```
// Lock two mutexes
std::lock(mut1, mut2);
```

Adopting Locks

- std::unique_lock can "adopt" the locks
  - Pass the std::adopt_lock option to its constructor
  - The std::unique_lock object now owns the lock

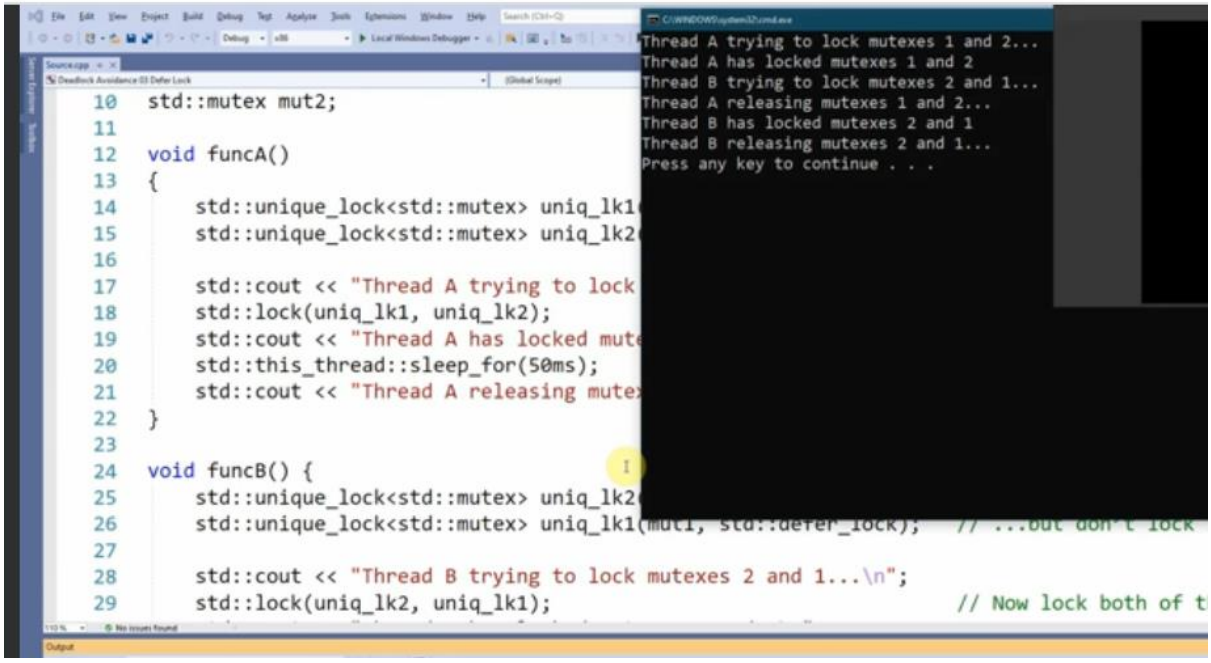
```
// Lock two mutexes
std::lock(mut1, mut2);

// Each lock is adopted by a unique_lock object
std::unique_lock<std::mutex> uniq_lk1(mut1, std::adopt_lock);
std::unique_lock<std::mutex> uniq_lk2(mut2, std::adopt_lock);
```



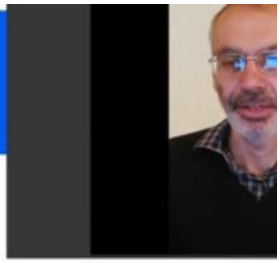
Deferring Locks

- Alternatively, we can can "defer" the locking
  - Pass the std::defer_lock option to the constructor
  - Then lock the mutexes later

```
// Each unique_lock object is associated with a mutex
std::unique_lock< std::mutex> uniq_lk1(mut1, std::defer_lock);
std::unique_lock< std::mutex> uniq_lk2(mut2, std::defer_lock);

// Lock the mutexes
std::lock(lk1, lk2);
```



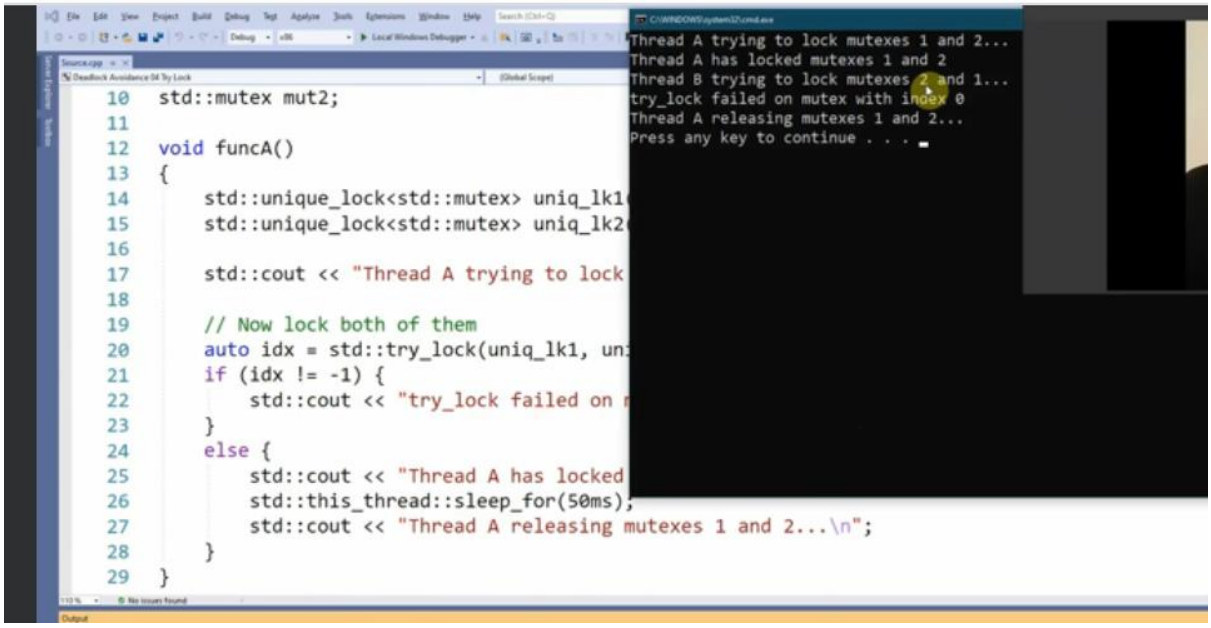
std::try_lock()



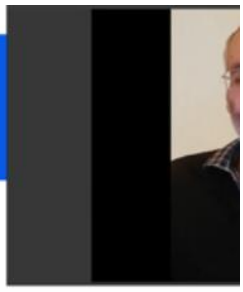
- Also locks multiple mutexes in a single operation

```
// Lock the mutexes
std::try_lock(uniq_lk1, uniq_lk2);
```

- Returns immediately if it cannot obtain all the locks
  - On failure, it returns the index of the object that failed to lock (0 for the first argument, etc)
  - On success, it returns -1



Hierarchical Mutex



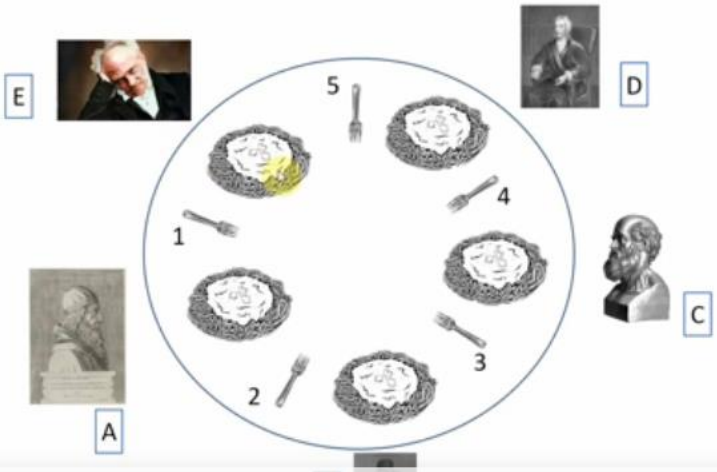
- Sometimes this approach is not suitable
  - It is not feasible to acquire multiple locks simultaneously
- A common technique is to impose an ordering
- A thread cannot lock a mutex unless it has already locked a mutex with a lower status
  - ID number
  - Alphabetical name
- The Williams book has a hierarchical_mutex that implements this

Deadlock Avoidance Guidelines

- Avoid waiting for a thread while holding a lock
  - The other thread may need the lock to proceed
- Try to avoid waiting for other threads
  - The other thread may be waiting for your thread
- Try to avoid nested locks
  - If your thread already holds a lock, do not acquire another one
  - If you need multiple locks, acquire them in a single operation
- Avoid calling functions within a critical section
  - Unless you are certain the function does not try to lock

Deadlock Avoidance Practical

Dining Philosophers

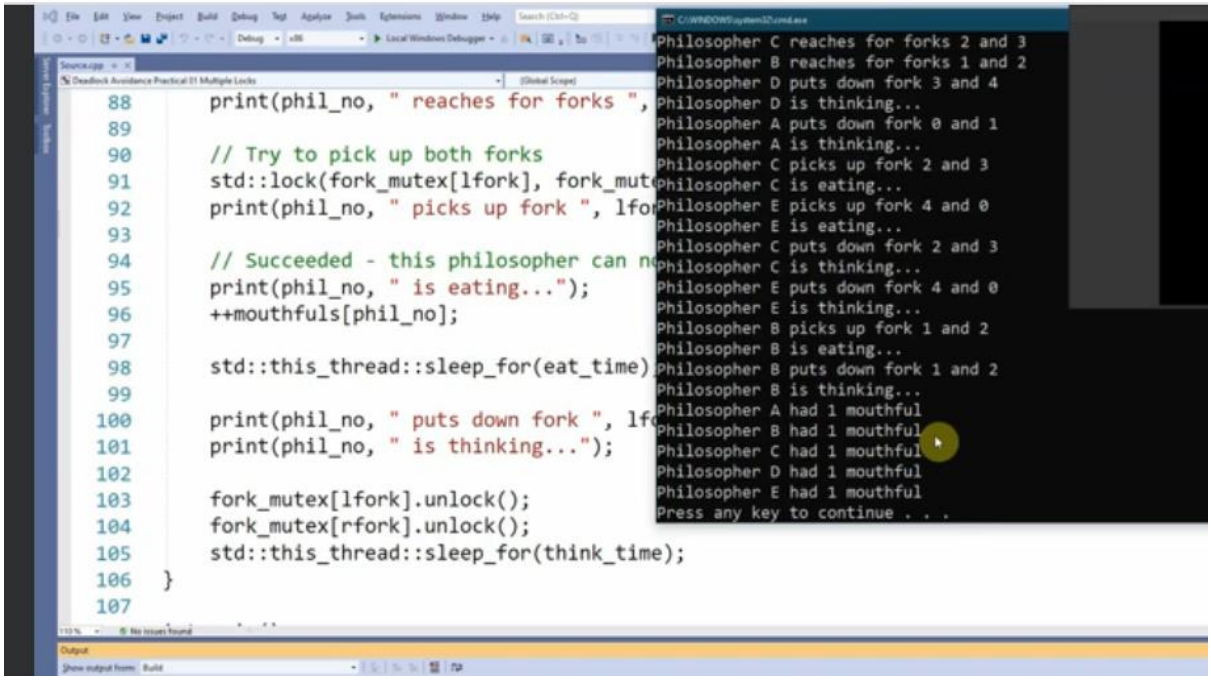


Deadlock Avoidance

- Use try_lock() with a timeout
  - Instead of blocking lock()
  - May result in livelock

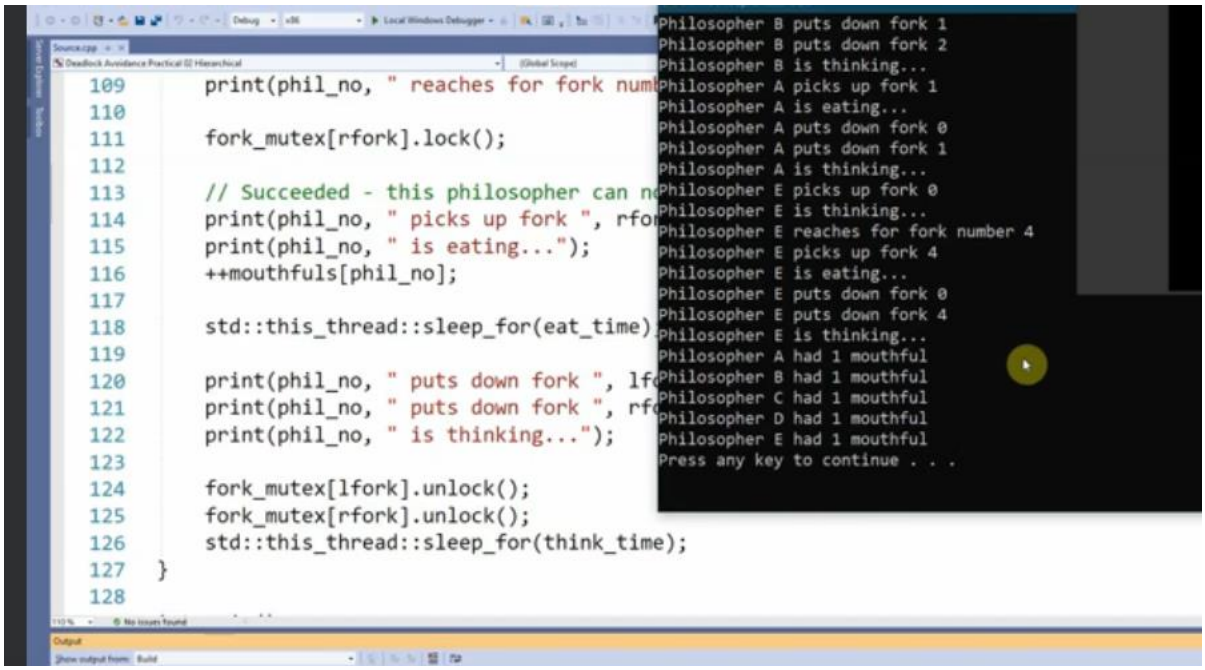
Deadlock Avoidance

- Use std::lock()
  - Lock both mutexes in a single operation



Deadlock Avoidance

- Use hierarchical ordering
  - Lock lower-numbered mutexes first



Livelock

Livelock

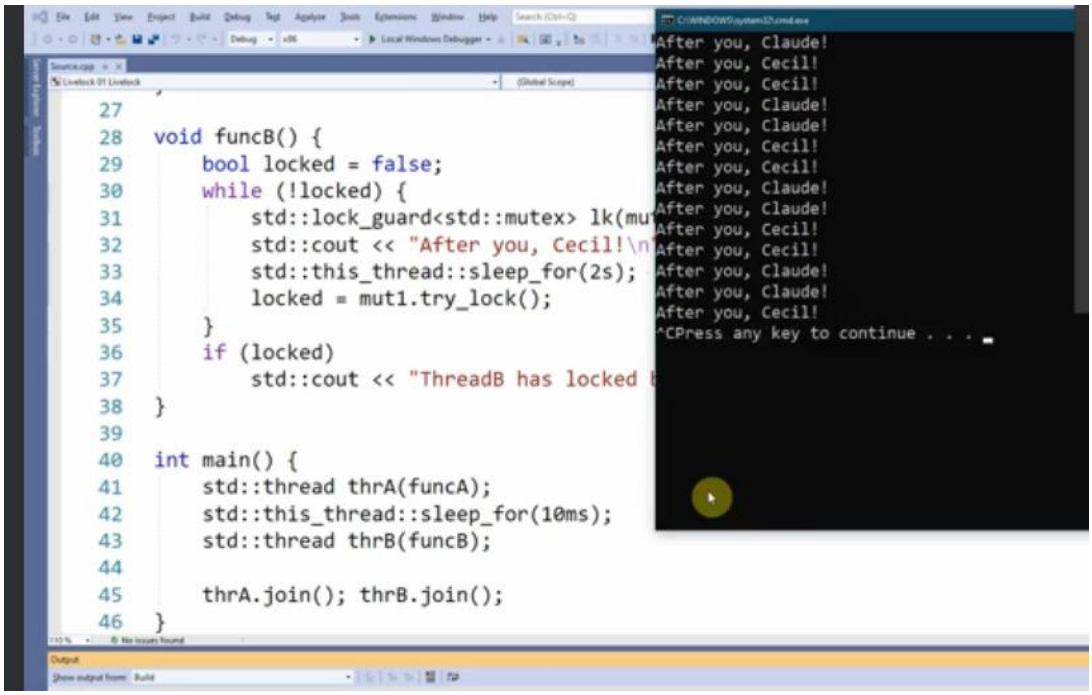
- A program cannot make progress
  - In deadlock, the threads are inactive
  - In livelock, the threads are still active
- Livelock can result from badly done deadlock avoidance
  - A thread cannot get a lock
  - Instead of blocking indefinitely, it backs off and tries again

Livelock Example

```
void funcA()
{
   bool locked = false;

   while (!locked) {
      std::lock_guard lck_guard(mut1);          // Lock mut1
      std::this_thread::sleep_for(1s);
      locked = mut2.try_lock();                 // Try to lock mut2
   }
}

void funcB()
{
   // Same as funcA, but with mut1 and mut2 interchanged
}
```

Livelock Analogy

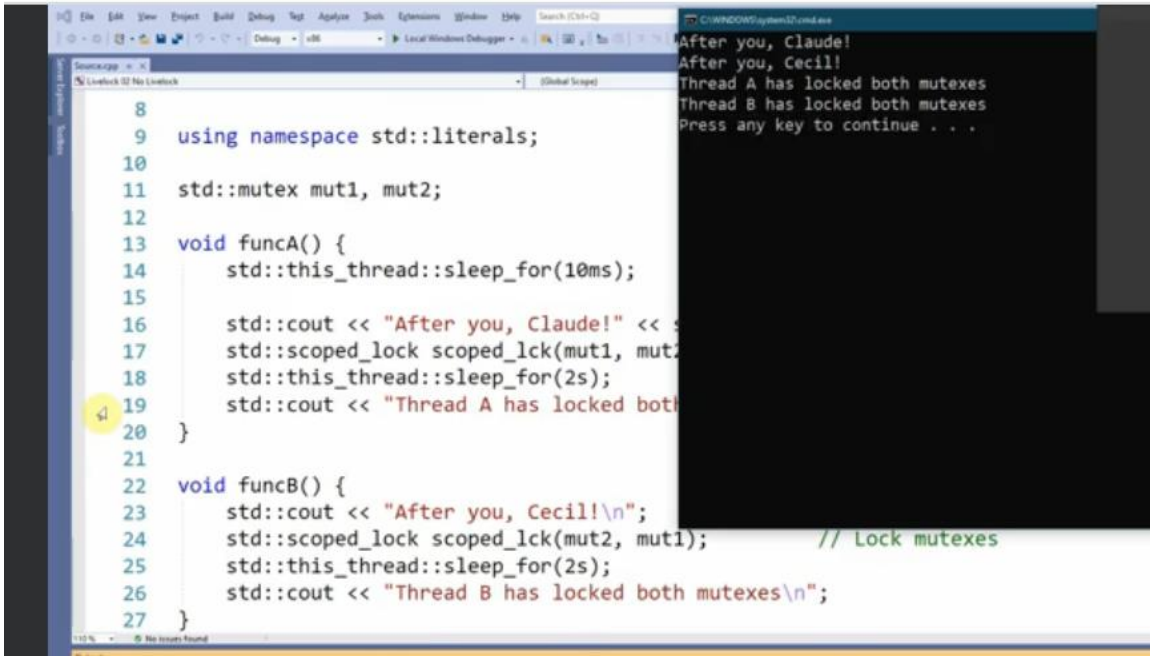
- Imagine two very polite people
- They walk down a corridor together
- They reach a narrow door
  - They each try to go through the door at the same time
  - Each one stops and waits for the other to go through the door
  - Then they both try to go through the door at the same time
  - Then each one stops and waits for the other to go through the door, etc



Livelock Avoidance

- Use std::scoped_lock or std::lock()
  - The thread can acquire multiple locks in a single operation
  - Built-in deadlock avoidance

```
void funcA()
{
  std::scoped_lock scoped_lck(mut1, mut2);          // Lock both mutexes
  // ...
}

void funcB()
{
  std::scoped_lock scoped_lck(mut2, mut1);          // Lock both mutexes
  // ...
}
```



Thread Priority

- We could assign different priorities to threads
- Not directly supported by C++
- Most thread implementations allow it
  - Accessible via std::thread's native_handle()
  - A high priority thread will run more often
  - A low priority thread will suspended or be interrupted more often
- The high priority thread will lock the mutex first
- The low priority thread will lock the mutex afterwards

Resource Starvation

- A thread cannot get the resources it needs to run
  - In deadlock and livelock, the thread cannot acquire a lock it
- Lack of system resources can prevent a thread starting
  - System memory exhausted
  - Maximum supported number of threads is already running
- Low priority threads may get starved of processor time
  - Higher priority threads are given preference by the scheduler
  - Good schedulers try to avoid this

Livelock Practical

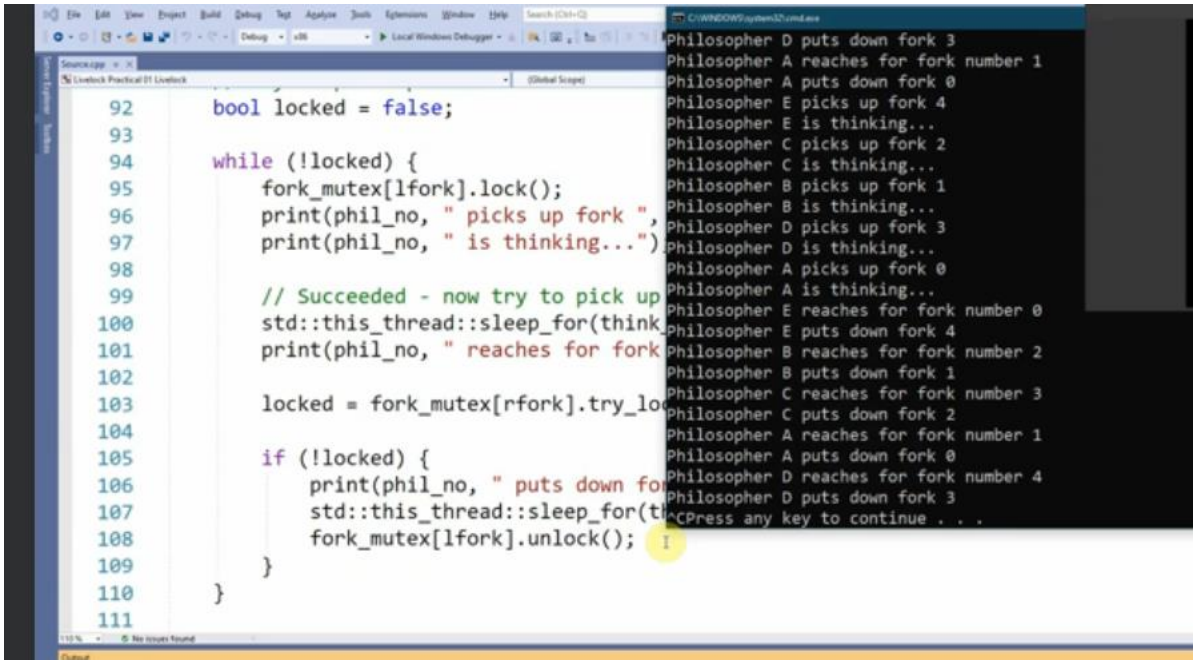
Dining Philosophers

Deadlock

- The previous attempt resulted in deadlock
  - All the philosophers picked up their left fork
  - None of the right forks were available
  - The philosophers were stuck in the "thinking" state

Deadlock Avoidance

- We try to avoid the deadlock
- Add a time-out and retry:
  - Philosopher picks up left fork
  - Philosopher tries to pick up right fork
  - Philosopher cannot pick up right fork
  - Philosopher puts down left fork
  - Philosopher waits
  - Philosopher picks up left fork again



Livelock

- This creates a situation of livelock:
  - All the philosophers pick up their left forks at the same time
  - All the philosophers try to pick up their right fork
  - All the philosophers put down their left forks at the same time
  - All the philosophers pick up their left forks at the same time
- The philosopher threads are livelocked
  - The philosophers are active, but cannot enter the "eating" state

Solutions

- Add randomness
  - The philosophers pick up and put down their forks at differe
  - Reduces the probability of starvation
  - Does not completely eliminate it
- Provide a central arbitrator to coordinate the philosophers
  - Only allows one philosopher to pick up a fork at a time
  - Only one philosopher can eat at a time
  - Reduces parallelism
- Use a shared lock
  - In effect, a philosopher picks up both forks at the same time

Solutions

- Introduce a fork hierarchy
  - The philosopher must pick up the lower-numbered fork first
  - A picks up fork 0
  - B picks up fork 1
  - C picks up fork 2
  - D picks up fork 3
  - E tries to pick up fork 0 (instead of 4)
- This leaves fork 4 available
  - D picks up fork 3
  - D starts eating