How do I destruct an object that keeps a reference to itself, because that reference prevents the object from being destructed

devblogs.microsoft.com/oldnewthing/20190306-00

March 6, 2019

Raymond Chen

Consider the case of an ordered work queue. Components can queue work to the worker thread, with the expectation that the work will run sequentially.

```
class OrderedWorkQueue
{
public:
 OrderedWorkQueue()
 {
  // Start up the worker thread.
  m_thread = std::thread([this]() {
   while (!m_exiting) {
    m_signal.Wait();
    m_queue.ProcessWork();
   }
  });
 }

 template<typename... Args>
 void QueueWork(Args&&... args)
 {
  m_queue.Append(std::make_unique<OrderedWorkItem>
                                  (std::forward<Args>(args)...));
  m_signal.Signal();
 }

 ~OrderedWorkQueue()
 {
  m_exiting = true;
  m_signal.Signal();
 }

private:
 // App-provided stuff. Assume they work.
 class OrderedWorkItem
 {
  template<typename... Args>
  OrderedWorkItem(Args&&... args);

  void Execute();
 };

 SomeSortOfQueue m_queue;
 SomeSortOfSignal m_signal;

 // Stuff related to worker thread management.
 std::atomic<bool> m_exiting;
 std::thread m_thread;
};
```

The `OrderedWorkQueue` object creates a worker thread whose job is to execute work items in the order they were queued. We have a problem here: The main thread signals the worker thread to exit, and then returns immediately, stranding the worker thread with a `this` pointer that points to a destructed object.

One solution is to wait for the worker thread to finish its work.

```
 ~OrderedWorkQueue()
 {
  m_exiting = true;
  m_signal.Signal();
  m_thread.join();
 }
```

This ensures that the `this` pointer remains valid for the duration of the worker thread. However, this comes with its own problems.

First, It changes the behavior of the class. destruction used to be a quick affair, but now destruction waits for the work items to drain, which can take an indefinite length of time. The intent of the `OrderedWorkQueue` may have been to employ a fire-and-forget design: Create an ordered work queue, queue a bunch of work to it, and then destroy the work queue. The work that got queued up will still execute eventually, in order, but the main thread was expecting to be able to get other work done in the meantime.

Furthermore, one of the work items may need to communicate with the thread that is doing the destructing, but it can't do that because the destructing thread is waiting for the worker thread to exit. So you have a potential deadlock.

Okay, so we solve this problem by having the worker thread maintain a strong reference to the object, to ensure that the object's member variables remain valid for the duration of the thread.

```
class OrderedWorkQueue
{
 static std::shared_ptr<OrderedWorkQueue> Create()
 {
  auto self = std::make_shared<OrderedWorkQueue>();
  self->m_thread = std::thread([lifetime = self, this]() {
   while (!m_exiting) {
    m_signal.Wait();
    m_queue.ProcessWork();
   }
  });
  return self;
 }
```

The captured `lifetime` retains the shared reference so that the background thread can continue using the object's member variables.

But wait, we have a new problem: The destructor never runs because the worker thread retains a strong reference to it.

Okay, so we try to fix the problem by passing a weak reference, and converting it to strong only as necessary.

```
static std::shared_ptr<OrderedWorkQueue> Create()
{
 auto self = std::make_shared<OrderedWorkQueue>();
 self->m_thread = std::thread(
  [weak = std::weak_ptr<OrderedWorkQueue>(self), this]() {
  auto strong = weak.lock();
  if (!strong) return;
  while (!m_exiting) {
   m_signal.Wait();
   auto workList = m_queue.DetachWork();
   // drop the strong reference while we process the work
   strong.reset();
   ProcessWorkList(workList);
   // reacquire the strong reference after work is done
   strong = weak.lock();
   if (!strong) return;
  }
 });
 return self;
}
```

This doesn't really go anywhere because the `m_signal.Wait()` call runs while there is still a strong reference, so we are back where we started.

One way out is to create a façade. The public-facing `OrderedWorkQueue` is what other components use to queue work to a background thread. The public-facing part retains a shared reference to the private part, and it's the private part that does the real work.

```
class OrderedWorkQueue
{
public:
 OrderedWorkQueue() = default;

 template<typename... Args>
 void QueueWork(Args&&... args)
 {
  m_worker->QueueWork(std::forward<Args>(args)...);
 }

 ~OrderedWorkQueue()
 {
  m_worker->Exit();
 }

private:
 // This is our old OrderWorkQueue class
 class OrderedWorkQueueWorker
 {
 public:
  static std::shared_ptr<OrderedWorkQueueWorker> Create()
  {
   auto self = std::make_shared<OrderedWorkQueueWorker>();
   self->m_thread = std::thread([lifetime = self, this]() {
    while (!m_exiting) {
     m_signal.Wait();
     m_queue.ProcessWork();
    }
   });
   return self;
  }

  template<typename... Args>
  void QueueWork(Args&&... args)
  {
   m_queue.Append(std::make_unique<OrderedWorkItem>
                                  (std::forward<Args>(args)...));
   m_signal.Signal();
  }

  void Exit()
  {
   m_exiting = true;
   m_signal.Signal();
  }

 private:
  // App-provided stuff. Assume they work.
  class OrderedWorkItem
  {
   template<typename... Args>
```

```
    OrderedWorkItem(Args&&... args);

    void Execute();
  };

  SomeSortOfQueue m_queue;
  SomeSortOfSignal m_signal;

  // Stuff related to worker thread management.
  std::atomic<bool> m_exiting;
  std::thread m_thread;
 };

 std::shared_ptr<OrderedWorkQueueWorker> m_worker =
                               OrderedWorkQueueWorker::Create();
};
```

An equivalent version which some people prefer is to put only the data members into the shared object.

```
class OrderedWorkQueue
{
public:
 OrderedWorkQueue()
 {
  m_thread = std::thread([data = m_data]() {
   while (!data->m_exiting) {
    data->m_signal.Wait();
    data->m_queue.ProcessWork();
   }
  });
 }

 template<typename... Args>
 void QueueWork(Args&&... args)
 {
  m_data->m_queue.Append(std::make_unique<OrderedWorkItem>
                                  (std::forward<Args>(args)...));
  m_data->m_signal.Signal();
 }

 ~OrderedWorkQueue()
 {
  m_data->m_exiting = true;
  m_data->m_signal.Signal();
 }

private:
 struct OrderWorkQueueData
 {
  SomeSortOfQueue m_queue;
  SomeSortOfSignal m_signal;
  std::atomic<bool> m_exiting;
 };

 class OrderedWorkItem
 {
  template<typename... Args>
  OrderedWorkItem(Args&&... args);

  void Execute();
 };

 std::shared_ptr<OrderedWorkQueueData> m_data =
   std::make_shared<OrderedWorkQueueData>();

 std::thread m_thread;
};
```

Raymond Chen

Follow