Shared Pointer Solutions

std::shared_ptr

- Briefly describe std::shared_ptr
  - std::shared_ptr is a "smart pointer" class
  - Unlike std::unique_ptr, a shared_ptr object can be copied and assigned to
  - This allows multiple shared_ptr objects to have access to the allocated memory
  - When the last shared_ptr object that accesses the shared memory is destroyed, the memory is released

std::shared_ptr Structure

- Briefly describe the structure of the shared_ptr class
  - shared_ptr has a member which is a pointer to the shared memory
  - It has another member which is a pointer to a "control block"
  - The control block contains a reference counter which keeps track of the number of objects that can access the shared memory
  - Every time one of these objects is copied or assigned from, the reference counter is incremented
  - Every time one of these objects is destroyed, the reference counter is decremented
  - When the last of the objects is destroyed, the counter goes to zero and the shared memory is released

std::shared_ptr Initialization

- Give two ways to create an initialized std::shared_ptr object
  - Call std::make_shared()
  - Pass a pointer as argument to the shared_ptr constructor
- Is there any reason to prefer one approach over the other?
  - make_shared() will make a single call to new() to allocate the shared memory and the control block in a single, contiguous location in memory
  - The constructor call will make a second call to new() to allocate the control block
  - This will probably have a different address from the first pointer, and the processor will not be able to optimize the data fetches into a single operation
  - std::make_shared() should be preferred as it avoids this extra overhead

Copying std::shared_ptr

- Describe what happens when a shared_ptr object is created as a copy of another object
  - The new object will share its pointer member and control block with those in the original object
  - The reference counter in the shared control block will be incremented

std::shared_ptr Operations

- Write a simple program which creates and initializes shared_ptr object and performs some operations on it

Threads and std::shared_ptr

- What issues arise when shared_ptr is used in a multithreaded program?
  - The reference counter is atomic
  - This prevents data races when a shared_ptr object is copied or assigned to
  - However, the data in the allocated memory requires protection if there are conflicting accesses to it
- What impact do these issues have on the performance of shared_ptr?
  - Using atomic operations on the reference counter adds a significant amount of overhead to shared_ptr
  - Normally, unique_ptr should be used instead, unless the extra features of shared_ptr are needed

Threads and std::shared_ptr

- Write a program which creates an std::shared_ptr object, then starts two threads
    - The threads copy the std::shared_ptr object concurrently
    - Make sure your program is thread-safe
- Write a program which creates an std::shared_ptr object, then starts two threads
    - The threads modify the data in the std::shared_ptr object concurrently
    - Make sure your program is thread-safe

Shared Pointer Applications

- Give an example where std::shared_ptr could be useful
  - In applications which have many copies of the same data in allocated memory, shared_ptr will save memory by only creating a single copy of the data. The other instances will be a reference to this single copy
  - A large document will contain many duplicated words: we can save memory by storing each of these words once in a shared_ptr
  - A web browser where several tabs are displaying the same image, such as a company logo: we can save memory by storing the image in a shared_ptr