CS 540 Programming Assignment 3

Due May 6th, 11:59 PM

All classes should be in the `cs540` namespace. Your code must work with test classes that are not in the `cs540` namespace, however. Your code should not have any memory errors or leaks as reported by `valgrind`. Your code should compile and run on the `remote.cs.binghamton.edu` cluster. Your code must not have any hard-coded, arbitrary limits or assumptions about maximum sizes, etc. Your code should compile without warning with `-Wall -Wextra -pedantic`. Special exemptions to this may be requested in advance.

Implement a non-intrusive, thread-safe, exception-safe, reference-counting smart pointer named `cs540::SharedPtr`. Our model for this will be `std::shared_ptr`, so you can read up on that to understand how it should behave.

`SharedPtr` must allow different smart pointers in different threads to be safely assigned and unassigned to the same shared objects. You may use either locks or atomic increment operations to ensure thread-safety. You do not need to (and in fact should not) ensure that the same `SharedPtr` can be used concurrently.

Note that when the smart pointer determines that the object should be deleted, it must delete the object via a pointer to the original type, even if the template argument of the final smart pointer is of a base type. This is to avoid object slicing for non-virtual destructors.

You may (and should) create any helper classes you feel necessary.

NOTE: We are using the term “object” as in the standard, where it refers to any region of storage except functions. Thus, these smart pointers should be able to work with fundamental types as well as objects of class type.

- Shared Pointer Test: Check the comment at the beginning of the file.

The API is given below.

Template

Declaration	Description
`template <typename T> class SharedPtr;`	The smart pointer points to an object of type *T*. *T* may refer to a `const`-qualified type.

Public Member Functions

Prototype	Description
Constructors, Assignment Operators, and Destructor	
`SharedPtr();`	Constructs a smart pointer that points to null.
`template <typename U> explicit SharedPtr(U *);`	Constructs a smart pointer that points to the given object. The reference count is initialized to one.

`SharedPtr(const SharedPtr &p);`
`template <typename U> SharedPtr(const SharedPtr<U> &p);`
	If *p* is not null, then reference count of the managed object is incremented. If `U *` is not implicitly convertible to `T *`, use of the second constructor will result in a compile-time error when the compiler attempts to instantiate the member template.

`SharedPtr(SharedPtr &&p);`
`template <typename U> SharedPtr(SharedPtr<U> &&p);`
	Move the managed object from the given smart pointer. The reference count must remain unchanged. After this function, *p* must be null. This must work if `U *` is implicitly convertible to `T *`.

`SharedPtr &operator=(const SharedPtr &);`
`template <typename U>`
`SharedPtr<T> &operator=(const SharedPtr<U> &);`
	Copy assignment. Must handle self assignment. Decrement reference count of current object, if any, and increment reference count of the given object. If `U *` is not implicitly convertible to `T *`, this will result in a syntax error. Note that both the normal assignment operator and a member template assignment operator must be provided for proper operation.

`SharedPtr &operator=(SharedPtr &&p);`
`template <typename U> SharedPtr &operator=(SharedPtr<U> &&p);`
	Move assignment. After this operation, *p* must be null. The reference count associated with the object (if *p* is not null before the operation) will remain the same after this operation. This must compile and run correctly if `U *` is implicitly convertible to `T *`, otherwise, it must be a syntax error.

`~SharedPtr();`	Decrement reference count of managed object. If the reference count is zero, delete the object.
Modifiers	
`void reset();`	The smart pointer is set to point to the null pointer. The reference count for the currently pointed to object, if any, is decremented.
`template <typename U> void reset(U *p);`	Replace owned resource with another pointer. If the owned resource has no other references, it is deleted. If *p* has been associated with some other smart pointer, the behavior is undefined.
Observers	

Prototype	Description
`T *get() const;`	Returns a pointer to the owned object. Note that this will be a pointer-to-`const` if *T* is a `const`-qualified type.
`T &operator*() const;`	A reference to the pointed-to object is returned. Note that this will be a `const`-reference if *T* is a `const`-qualified type.
`T *operator->() const;`	The pointer is returned. Note that this will be a pointer-to-`const` if *T* is a `const`-qualified type.
`explicit operator bool() const;`	Returns true if the `SharedPtr` is not null.

Non-member (Free) Functions

Prototype	Description

`template <typename T1, typename T2>`
`bool operator==(const SharedPtr<T1> &, const SharedPtr<T2> &);`
	Returns true if the two smart pointers point to the same object or if they are both null. Note that implicit conversions may be applied.

`template <typename T>`
`bool operator==(const SharedPtr<T> &, std::nullptr_t);`
`template <typename T>`
`bool operator==(std::nullptr_t, const SharedPtr<T> &);`
	Compare the `SharedPtr` against `nullptr`.

`template <typename T1, typename T2>`
`bool operator!=(const SharedPtr<T1>&, const SharedPtr<T2> &)`
	True if the `SharedPtr`s point to different objects, or one points to null while the other does not.

`template <typename T>`
`bool operator!=(const SharedPtr<T> &, std::nullptr_t);`
`template <typename T>`
`bool operator!=(std::nullptr_t, const SharedPtr<T> &);`
	Compare the `SharedPtr` against `nullptr`.

`template <typename T, typename U>`
`SharedPtr<T> static_pointer_cast(const SharedPtr<U> &sp);`
	Convert *sp* by using `static_cast` to cast the contained pointer. It will result in a syntax error if `static_cast` cannot be applied to the relevant types.

`template <typename T, typename U>`
`SharedPtr<T> dynamic_pointer_cast(const SharedPtr<U> &sp);`
	Convert *sp* by using `dynamic_cast` to cast the contained pointer. It will result in a syntax error if `dynamic_cast` cannot be applied to the relevant types, and will result in a smart pointer to null if the dynamic type of the pointer in *sp* is not *T* *.