

Assignment 8 - Smart Pointers

[Submit Assignment](#)

Due Tuesday by 11:59pm **Points** 32 **Submitting** a file upload **File Types** hpp
Available Nov 10 at 12am - Nov 21 at 11:59pm 12 days

The purpose of this assignment is to give you another experience in developing C++ template code. You'll gain specific experience in developing destructors, move constructor, and move assignment operations, in addition to raw memory management.

The subject of this assignment is the creation of a `usu::shared_ptr` class that provides smart pointer capabilities just like the `std::shared_ptr`.

The `usu::shared_ptr` class is patterned after the `std::shared_ptr` class, providing almost all of the same capabilities, while also adding the ability to manage arrays!

For our purposes: A shared pointer works by creating a (raw) pointer to the object/primitive for which it is managing the memory. It also creates a (raw) pointer to an unsigned integer that keeps track of the number of active references to the memory. As copies of the shared pointer are made, these pointers need to be managed by the shared pointer code, updating, copying, and/or moving them as appropriate. When a shared pointer goes out of scope, the destructor is invoked, and logic written to update the reference count and clean up memory if it is the last reference to the allocated memory.

This assignment will challenge you to think carefully not only about writing generic code, but also about how a smart pointer works. There isn't a lot of code, it is creating the "right" code that is crucial to this assignment.

Assignment

Write a templated `shared_ptr` class, contained with a `usu` namespace according to the following specifications.

- Constructors
 - Overloaded that takes a `T*` : Stores the raw pointer and initializes the reference count to 1.
 - Copy constructor : Make a copy of the `shared_ptr` and increments the reference count.
 - Move constructor : Moves the `shared_ptr`, does not increment the reference count.
- Destructor : decrements the reference count. If the reference count goes to 0, cleans up any allocated memory.
- Operators
 - assignment operator : Copies the `shared_ptr` into the destination, increments the reference count.
 - move assignment operator : Moves the `shared_ptr` into the destination, does not increment the reference count.
 - pointer operator (`->`) : Returns a pointer to the managed raw pointer.
 - dereference operator (`*`) : Dereferences the managed raw pointer, returning the value at the memory

location.

- Other methods
 - `get` : Returns a pointer to the managed raw pointer.
 - `use_count` : Returns the reference count.

If that isn't enough, make another `shared_ptr` class for arrays! A few differences between this class and the previous one.

- The type declaration will start like...
 - `template<typename T>`
`class shared_ptr<T[]>`
- The overloaded constructor accepts two parameters:
 1. Raw pointer to the array.
 2. The number of elements in the array.
- The destructor needs to delete an array.
- Overload the `[]` operator for array-like access.
- A `size` method that returns the number of elements in the array.
- Does not implement the `->` operator, the `*` operator or the `get` method.

Example Usage & Unit Tests

An example `main.cpp` is provided that exercises some of the capabilities of the code you need to write. Additionally you are provided a set of unit tests that exercise the capabilities of the smart pointer classes.

The last section of this assignment description shows the output from running the `main.cpp` code over my implementation of the smart pointer code.

Files

- [main.cpp](#) 
- [TestMemory.cpp](#) 

Implementation & Other Notes

It goes without saying, but I'm going to say it anyway, you must use only raw pointers in your smart pointer code implementation.

The `make_shared` function is a little tricky to write and explain. Because I know there will be a lot of questions on how to write it, here it is...

```
template <typename T, typename... Args>
shared_ptr<T> make_shared(Args&&... args)
{
    return shared_ptr<T>(new T(std::forward<Args>(args)...));
}
```



This function uses variadic templates to forward l-values and r-values appropriately, based on the type of `T`. In other words, it correctly passes arguments to the constructor of type `T`, based on the parameters passed to the function.

Note this function performs the raw pointer memory allocation and passes that raw pointer into the `shared_ptr` constructor.

Here is the corresponding one for creating a shared array pointer...

```
template <typename T, unsigned int N>
shared_ptr<T[]> make_shared_array()
{
    return shared_ptr<T[]>(new T[N], N);
}
```

Submission Notes

- Turn in only the following file: **shared_ptr.hpp**
 - Both shared pointer classes, and provided functions, are placed in this single file.
- Your code must compile without any warnings or compiler errors; against the provided [main.cpp](#)  and [TestMemory.cpp](#)  provided code. See syllabus regarding code that has compiler errors.
- Your code must adhere to the CS 3460 coding standard: [link](#)
- Your code must be formatted through the use of the following clang-format configuration file: [link](#)

Example Run

This is most definitely not interesting output, but shows what you should expect from the driver code.

```
--- Reference Counts ---
p13 : 1
P14 : 1
auto p13b = p13
auto p14b = p14
p13 : 2
P14 : 2
auto p13b = p11
auto p14b = p12
p11 : 2
P12 : 2
p13b : 2
P14b : 2
p13 : 1
P14 : 1

--- Member Access ---
from p13
from p14
from (*p13)
from (*p14)
from p13.get()
from p14.get()
this is a test
this is a test
this is a test
this is a test
```

Assignment 8 - Smart Pointers (1)

Criteria	Ratings		Pts
Compiles without any warnings (MSVC and g++). on Linux (-Wall -Wextra -pedantic) on Windows (/W4 /permissive-)	2.0 pts Full Marks	0.0 pts No Marks	2.0 pts
Follows required course code style. Must also be formatted using clang-format.	2.0 pts Full Marks	0.0 pts No Marks	2.0 pts
Passes all unit tests & matches sample output	4.0 pts Full Marks	0.0 pts No Marks	4.0 pts
Valid constructors Both shared_ptr types: Overloaded, Copy, and Move	6.0 pts Full Marks	0.0 pts No Marks	6.0 pts
Correct reference counting and memory cleanup Both shared_ptr types: This includes management of the raw pointer to the data and the raw pointer used to store the reference count.	6.0 pts Full Marks	0.0 pts No Marks	6.0 pts
Correct assignment and move assignment operators Both shared_ptr types	4.0 pts Full Marks	0.0 pts No Marks	4.0 pts
pointer -> operator Shared pointer only	1.0 pts Full Marks	0.0 pts No Marks	1.0 pts
Dereference operator * Shared pointer only	1.0 pts Full Marks	0.0 pts No Marks	1.0 pts
use_count function Both shared_ptr types	2.0 pts Full Marks	0.0 pts No Marks	2.0 pts
Array access operator Shared array only	2.0 pts Full	0.0 pts No	2.0 pts

Criteria	Ratings		Pts
	Marks	Marks	
size method Shared array only	2.0 pts Full Marks	0.0 pts No Marks	2.0 pts
Total Points: 32.0			

