CS 540 Programming Assignment 3: Templates

Due December 12th, 2014

(This document was last modified on Monday, December 1, 2014 at 01:32:00 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.

Description

Part 1: Smart Pointer (50 pts)

Implement a smart pointer similar to `std::shared_ptr` that only deletes its managed object when the associated reference count reaches 0. `SharedPtr` should be copyable and assignable. Such an operation should increase the reference count of the associated object, not duplicate the object pointed to.

`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.

Your smart pointer should be non-intrusive. By non-intrusive, we mean that it must work without requiring that the pointed-to class be modified.

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

All functions given below are required.

Prototype	Description
Constructors	
`SharedPtr()`	Construct a `SharedPtr` to `nullptr`
`template<typename U> explicit SharedPtr(U *)`	Construct a `SharedPtr` with the given object, with a reference count of 1
`SharedPtr(const SharedPtr&sp)`	Duplicates a `SharedPtr`. If sp points to something, then reference count of the managed object is incremented
`template <typename U> SharedPtr(const SharedPtr<U>&)`	This must work if `U*` is implicitly convertible to `T*`
`SharedPtr(SharedPtr&&)`	Takes resource from another `SharedPtr`, the reference count must remain unchanged and the other `SharedPtr` must lose its reference to the managed object
`template <typename U> SharedPtr(SharedPtr<U>&&)`	This must work if `U*` is implicitly convertible to `T*`
`~SharedPtr()`	Decrement reference count of managed object. If the reference count is zero, delete the object.
Member Functions	
`T *get() const`	Returns a pointer to the owned object
`template <typename U> void reset(U *p)`	Replace owned resource with another pointer. If the owned resource has no other references,

Prototype	Description
	it is deleted. If *p* has been associated with some other smart pointer, the behavior is undefined.
`void reset(std::nullptr_t)`	Lose reference to the previously managed object
Operators	
`SharedPtr& operator=(SharedPtr&&)`	Move assignment. Decrement reference count of managed resource and get reference to the other resource. The other `SharedPtr` must lose its reference to the managed object. The reference count associated with the other resource must not be changed
`template <typename U> SharedPtr& operator=(SharedPtr<U>&&)`	This must work if `U*` is implicitly convertible to `T*`
`SharedPtr& operator=(const SharedPtr&)`	Copy assignment. Must handle self assignment. Decrement reference count of managed resource and get reference count to other resource. Increment reference count associated with other resource
`template <typename U> SharedPtr& operator=(const SharedPtr<U>&)`	This must work if `U*` is implicitly convertible to `T*`
`T& operator*() const`	Return a reference to the owned object.
`T *operator->() const`	Returns a pointer to the owned object for access through structure dereference
`explicit operator bool() const`	Returns true if the `SharedPtr` owns a non-nullptr address
Comparison Operators (Non-Member Functions)	
`template <typename T1, typename T2>` `bool operator==(const SharedPtr<T1>&, const SharedPtr<T2>&)`	True if both `SharedPtr`s hold the same address
`template <typename T1>` `bool operator==(const SharedPtr<T1> &, std::nullptr_t)`	Compare `SharedPtr` with `nullptr`
`template <typename T1>` `bool operator==(std::nullptr_t, const SharedPtr<T1> &)`	Compare `SharedPtr` with `nullptr`
`template <typename T1, typename T2>` `bool operator!=(const SharedPtr<T1>&, const SharedPtr<T2> &)`	True if the `SharedPtr`s hold different addresses
`template <typename T1>` `bool operator!=(const SharedPtr<T1> &, std::nullptr_t)`	Compare SharedPtr with nullptr
`template <typename T1>` `bool operator!=(std::nullptr_t, const SharedPtr<T1> &)`	Compare SharedPtr with nullptr

Part 2: Arbitrary Dimension Array Class Template (50 pts)

For this part, you will implement a array class template named `cs540::Array` that can be instantiated with any number of dimensions. The size of dimensions is given as a sequence of `size_t` constant template arguments.

Here is an example of how it might be used. The full API is further below.

```
#include "Array.hpp"

#pragma GCC diagnostic push
#pragma GCC diagnostic ignored "-Wunused-variable"
int
main() {
    // Define a 2X3X4 array of integers.  Elements are uninitialized.
    cs540::Array<int, 2, 3, 4> a, b;
    cs540::Array<short, 2, 3, 4> c;
    // cs540::Array<int, 0> e1; // This line must cause a compile-time error.
    // Initialize.
    int counter = 0;
    for (int i = 0; i < 2; i++) {
        for (int j = 0; j < 3; j++) {
            for (int k = 0; k < 4; k++) {
                a[i][j][k] = b[i][j][k] = counter;
                c[i][j][k] = counter;
                counter++;
            }
        }
    }

    // Range-checked indexing.
    a[0][0][0] = 1234;
    a[1][1][1] = a[0][0][0];
    a[0][2][3] = 5678; // Set the element in the first plane, 3rd row, and 4th column.
    try {
        a[0][3][0] = 1; // Out of range, throws.
        assert(false);
    } catch (cs540::OutOfRange) {
    }

    a = a; // Self-assignment must be a no-op.
    b = a; // Okay, as long as dimensions and type match exactly.
    a = c; // Member template constructor.

    const cs540::Array<int, 2, 3, 4> &const_ref(a);

    int i = const_ref[0][0][0]; // Okay.
    // const_ref[0][0][0] = 1; // Syntax error.

    // Iterate through in plane major order.  (Note: Normally, you would use
    // auto to simplify.)
    std::cout << "First dimension major (row-major in 2-D): " << std::endl;
    for (cs540::Array<int, 2, 3, 4>::FirstDimensionMajorIterator it = a.fmbegin(); it != a.fmend(); ++it) {
        std::cout << *it << std::endl;
    }

    // Iterate through in column major order.
    std::cout << "Last dimension major (column-major in 2-D): " << std::endl;
    for (cs540::Array<int, 2, 3, 4>::LastDimensionMajorIterator it = a.lmbegin(); it != a.lmend(); ++it) {
        std::cout << *it << std::endl;
    }
}
#pragma GCC diagnostic pop
```

You should throw an object of class type `cs540::OutOfRange` as an exception if there is indexing operation that is out of range. You should provide the definition of this class in your header file.

Your code must not have any fixed limits. You must be able to construct arrays with any number of elements and dimensions, etc.

Your array implementation must be completely contained in `Array.hpp` and `Array.cpp`. Note that `Array.cpp` will likely be empty. Test code for your array is here. The correct output is here. Your code must work with the test code without change. We reserve the right to add addtional tests to this as we see fit, but we will conform to the API used in the provided test code.

Template

Declaration	Description
`template <typename T, size_t... Dims> class Array;`	This declares a multidimensional array containing elements of type *T* with the given dimensions. If any dimension is zero, it should fail to compile. Use `static_assert` to force this behavior.

Type Members

Member	Description
`ValueType`	The type of the elements: *T*.
`FirstDimensionMajorIterator`	The type of the named iterator.
`LastDimensionMajorIterator`	The type of the named iterator.

Public Member Functions

Prototype	Description
`Array();`	The default constructor must be defined, either explicitly or implicitly.
`Array(const Array &);`	The copy constructor must work. The dimensionality of the source array must be the same. Note that a non-template copy constructor must be provided, in addtion to the member template copy constructor.

`template <typename U>`
`Array(const Array<U, Dims...> &);`
	The copy constructor must work. The dimensionality of the source array must be the same.

Prototype	Description
`Array &operator=(const Array &);`	The assignment operator must work. The dimensionality of the source array must be the same. Self-assignment must be a no-op. Note that this non-template assignment operator must be provided, in addtion to the member template assignment operator.
`template <typename U>` `Array &operator=(const Array<U, Dims...> &);`	The assignment operator must work. The dimensionality of the source array must be the same. Self-assignment must be a no-op.
`T &operator[size_t i_1][size_t i_2]...[size_t i_D];` `const T &operator[size_t i_1][size_t i_2]...[size_t i_D] const;`	This is used to index into the array with range-checking. If any given indices are out-of-range, then an `OutOfRange` exception must be thrown. Note that this is a "conceptual" operator only. Such an operator does not really exist. Instead, you must figure out how to provide this functionality. (Helper classes may be useful.)
Iterators	
`FirstDimensionMajorIterator fmbegin();`	Returns a `FirstDimensionMajorIterator` (nested classes) object pointing to the first element.
`FirstDimensionMajorIterator fmend();`	Returns a `FirstDimensionMajorIterator` (nested classes) object pointing one past the last element.
`LastDimensionMajorIterator lmbegin();`	Returns a `LastDimensionMajorIterator` pointing to the first element.
`LastDimensionMajorIterator lmend();`	Returns a `LastDimensionMajorIterator` pointing one past the last element.

Public Member Functions for Nested Class `FirstDimensionMajorIterator`

This iterator is used to iterate through the array in row-major order. This iterator can be be used to read or write from the array.

Prototype	Description
`FirstDimensionMajorIterator()`	Must have correct default ctor.
`FirstDimensionMajorIterator(const FirstDimensionMajorIterator &)`	Must have correct copy constructor. (The implicit one will probably be correct.)
`FirstDimensionMajorIterator &` `operator=(const FirstDimensionMajorIterator &)`	Must have correct assignment operator. (The implicit one will probably be correct.)
`bool operator==(const FirstDimensionMajorIterator &,` `const FirstDimensionMajorIterator &)`	Compares the two iterators for equality.
`bool operator!=(const FirstDimensionMajorIterator &,` `const FirstDimensionMajorIterator &)`	Compares the two iterators for inequality.
`FirstDimensionMajorIterator &operator++()`	Increments the iterator one element in row-major order, and returns the incremented iterator (preincrement). If the iterator is pointing to the end of the list, the behavior is undefined.
`FirstDimensionMajorIterator operator++(int)`	Increments the iterator one element in row-major, and returns an iterator pointing to the element prior to incrementing the iterator (postincrement). If the iterator is pointing to the end of the list, the behavior is undefined.
`T &operator*() const`	Returns a reference to the *T* at this position in the array. If the iterator is pointing to the end of the list, the behavior is undefined.

[Extra Credit] Public Member Functions for Nested Class `LastDimensionMajorIterator`

This iterator is used to iterate through the array in column-major order. This iterator can be only be used to read or write from the array.

Prototype	Description
`LastDimensionMajorIterator()`	Must have correct default constructor.
`LastDimensionMajorIterator(const LastDimensionMajorIterator &)`	Must have correct copy constructor. (The implicit one will probably be correct.)
`LastDimensionMajorIterator & operator=(const LastDimensionMajorIterator &)`	Must have correct assignment operator. (The implicit one will probably be correct.)
`bool operator==(const LastDimensionMajorIterator &, const LastDimensionMajorIterator &)`	Compares the two iterators for equality.
`bool operator!=(const LastDimensionMajorIterator &, const LastDimensionMajorIterator &)`	Compares the two iterators for inequality.
`LastDimensionMajorIterator &operator++()`	Increments the iterator one element in column-major order, and returns the incremented iterator (preincrement). If the iterator is pointing to the end of the list, the behavior is undefined.
`LastDimensionMajorIterator operator++(int)`	Increments the iterator one element in column-major order, and returns an iterator pointing to the element prior to incrementing the iterator (postincrement). If the iterator is pointing to the end of the list, the behavior is undefined.
T `&operator*() const`	Returns a reference to the *T* at this position in the array. If the iterator is pointing to the end of the list, the behavior is undefined.

Part 3 [Extra Credit]: Interpolated `ostream` Output (30 pts)

The C++ `<iostream>` is arguably more idiomatic for C++, but is harder to read and type (at least for me) than C's `printf()`. This is an attempt to get the best of both worlds.

Write a function template named `Interpolate` that will make the below work. Each argument will be output when its corresponding `%` is encountered in the format string. All output should be ultimately done with the appropriate overloaded `<<` operator. A `\%` sequence should output a percent sign.

```
SomeArbitraryClass obj;
int i = 1234;
double x = 3.14;
std::string str("foo");
std::cout << Interpolate(R"(i=%, x1=%, x2=%\%, str1=%, str2=%, obj=%)", i, x, 1.001, str, "hello", obj) << std::endl;
```

If there is a mismatch between the number of percent signs and the number of arguments to output, throw an exception of type `cs540::WrongNumberOfArgs`.

Support the I/O manipulators listed here. An I/O manipulator should not "consume" a percent sign. You should be able to support these without creating a partial specialization or overloaded function template for every single manipulator.

A test program is here. Compile with `-DCS540_TEST_MANIPS` to test the I/O manipulators.