# CS 311 Fall 2023
# Assignment 5

Assignment 5 is due at **5 pm Thursday, November 2**. It is worth 65 points.

## Procedures

This is, optionally, a **group assignment**. You may work in a group of **two**, if you wish. Each group only needs to turn in a single copy of the assignment. Under normal circumstances, each group member will receive the same grade on the assignment.

Turn in answers to the exercises below on the UA Blackboard site, under Assignment 5 for this class.

- Your answers should consist of the source code for Exercise A (file `tmsarray.hpp`). This should be **attached** to your submission.
- I may not look at your homework submission immediately. If you have questions, e-mail me.

If you work in a group:

- The assignment files should contain the names of both group members.
- **One student** should submit the assignment as usual, on Blackboard.
- **The other student** should submit an empty assignment (no attached files) on Blackboard, with a note in the comments box indicating **whom** the assignment was done with.

## Exercises (65 pts total)

### Exercise A — Marvelously Smart Array Class Template

#### Purpose

In this exercise, you will write a class template that acts as a "smart array". It will be significantly better than the array class from Assignment 2—and almost as smart as `std::vector`. In particular, the array will be resizable and exception-safe, as well as efficient.

Key to this assignment is exception safety. Make sure that exceptions thrown by value-type operations are properly handled, and that all safety guarantees are documented.

And as always, make your code high quality.

#### Instructions

This assignment may be done individually or in a group of two.

Implement a C++ class template that manages and allows access to a resizable array. The type of item in the array should be specified by the client code. Be sure to follow the coding standards. **All standards now apply!**

- Name your class template "`TMSArray`", so that, for example, a `TMSArray` whose items have type `double` would be declared as `TMSArray<double>`.
- Implement your class template in file `tmsarray.hpp`. Since this is a template, there should be no associated source file.

Other than the class name and the fact that it is a class template with a client-specified value type, the interface for `TMSArray` should be that covered in recent class meetings. This interface is as follows.

- An object of type `TMSArray<Val>` should act as if it maintains a single array with a given size and value type `Val`.
- A const `TMSArray` should be one that does not allow modification of the items in its array. A non-const `TMSArray` should allow such modification.
- You may not create any new implicit type conversions. *Hint: 1-parameter constructors, "*`explicit`*".*
- Include the following **public member types**, and no others, in your class:
  - `value_type`. The type of each item in the array.
  - `size_type`. The type of the size of an array and an index into an array.
  - `iterator`. The type of a random-access iterator that allows modification of the item it references.
  - `const_iterator`. The type of a random-access iterator that does not allow modification of the item it references.
- Include the following **public member functions and operators**, and no others, in your class. *Note. "Integer" below does not necessarily mean* `int`*.*
  - Default ctor: creates a `TMSArray` of size zero.
  - Copy ctor, move ctor, dctor, copy assignment, move assignment operator.
    - The copy operations should create an entirely new copy of the data, so that modifying the copy does not change the original.
    - The move operations should have a similar apparent effect, but they may modify the original object, as long as they leave it in a valid state.
    - The dctor and the move operations must be noexcept.
  - Ctor from size. One parameter: a non-negative integer giving the number of items in the `TMSArray`.
  - Bracket operator. One parameter: a non-negative integer index from 0 to *size*–1, where *size* is the number of items in the array. Returns a reference to the proper item (const or non-const as appropriate).
  - `size`. No parameters. Returns a non-negative integer: the number of items in the array.
  - `empty`. No parameters. Returns a boolean value indicating whether the array has size zero.
  - `resize`. One parameter: a non-negative integer specifying the new size. Resizes the array, maintaining the values of all data items that lie in both the old and the new array. May need to reallocate-and-copy, but only does so when necessary.
  - `insert`. Two parameters: an iterator and a data item. Inserts the data item just *before* that referenced by the iterator—or at the end, if the passed iterator is the end iterator. Returns an iterator to the inserted item.
  - `erase`. One parameter: an iterator. Removes the item referenced by the iterator. Returns an iterator to the item just after the removed item—or `end()` if the removed item was the last.
  - `begin`. No parameters. Returns an iterator to item 0 in the array—or the same as `end()`, if the array has size 0.
  - `end`. No parameters. Returns an iterator to one-past the end of the array.

    - `push_back`. One parameter: an item of the value type. Appends this item to the end of the array, increasing the size of the array by 1. Nothing is returned.
    - `pop_back`. No parameters. Removes the last item from the array, decreasing the size of the array by 1. Nothing is returned.
    - `swap`. One parameter: another `TMSArray` having the same value type. Declared noexcept. Swaps the values of `*this` and the parameter. Must be constant-time.
- There should be no associated **global functions or operators** in your package.

In addition:

- You may not use any C++ Standard Library **classes** or **class templates** in your implementation. So no `std::vector`, and no `std::unique_ptr`. You may use simple types, like `std::size_t` and `std::ptrdiff_t`, and functions & function templates, like `std::swap`, `std::iter_swap`, `std::sort`, `std::rotate`, `std::fill`, `std::copy`, and `std::move`.
- All public functions should have the highest *reasonable* levels of efficiency and exception safety. In particular:
    - All public functions must offer at least the Basic Guarantee.
    - Where it is necessary to offer the No-Throw Guarantee, it must be offered.
    - The move constructor and the copy assignment and move assignment operators must be written in the exception-safe manner covered in class. (See the lecture slides for *Invisible Functions II* and *Exception Safety*.)
- All public functions must be exception-neutral; that is, exceptions thrown by value-type operations must propagate unchanged to the caller.
- Insert-at-end, using either `insert` or `push_back`, must be amortized constant-time.
- You may not forbid any member function of the value type from throwing, except for the destructor, move constructor, and move assignment.
- You must not use the catch-all-re-throw idiom unless there is actually clean-up that needs to be done.

### Test Program

A test program is available: `tmsarray_test.cpp`. If you compile and run the test program (unmodified!) with your code, then it will test whether your code works properly.

The test program requires `doctest.h`, the header for the *doctest* unit-testing framework, version 2.

Do not turn in the test program or the *doctest* framework.

### Notes

- Starting from class `MSArray` (in the Git repository) would be a good idea. Be sure the files you start from are marked "`VERSION 6`" or later. Remember, however, that `TMSArray` is a template, has no separate source file, will need a rather different copy constructor from the one `MSArray` has, and may require different parameter-passing methods, since array items may be large objects.
- Think about possible error conditions. What if the client calls `pop_back` on an empty `TMSArray`? What if the client calls `erase`, passing an iterator that does not point to an item in this container? Similar problems can occur in `insert` and the bracket operator. **What are you doing to do about it?**

- Going from a value type of `int`, as in `MSArray`, to an arbitrary value type, means that there are now more possible sources of error conditions. Doing anything with the value type, other than moving and destruction, might throw. Check every single possible source of errors, and make sure you deal with it appropriately.
- **Every class** needs class invariants.
- **Every function** needs documentation of the exception-safety guarantee it offers, and whether it is exception-neutral. (*They are all supposed to be exception-neutral.*) Those that have preconditions—other than class invariants—will need documentation of these.
- Ideas can be found in "*Thoughts on Assignment 5*", in the lecture slides.