ECS 36C: Programming Assignment #1

Instructor: Aaron Kaloti

Winter Quarter 2022

Contents

1 Changelog **1**

2 General Submission Details **1**

3 Purpose of This Assignment **2**

4 Reference Environment **2**

5 Programming Problems **2**
5.1 File Organization 2

6 Templated Array **2**
6.1 Static Assertions 2
6.2 Goal: Implementation of Templated Array 3
6.3 Restrictions 3
6.4 Examples 4

7 Grading Breakdown **4**

8 Autograder Details **5**

1 Changelog

You should always refer to the latest version of this document.

- v.1: Initial version.
- v.2:
  - Autograder details.
  - Mentioned that – if you wish – you could write your entire implementation in `array.hpp` and not submit `array.inl`.
- v.3: Fixed a typo that originally said you are to submit three files instead of two files.
- v.4: Pushed deadline back to the night of Wednesday, 01/19.

2 General Submission Details

Partnering on this assignment is prohibited. If you have not already, you should read the section on academic misconduct in the syllabus.

This assignment is due the night of Wednesday, 01/19. Gradescope will say 12:30 AM on Thursday, 01/20, due to the "grace period" (as described in the syllabus). *Be careful about relying on the grace period for extra time; this could be risky.*

*This content is protected and may not be shared, uploaded, or distributed.

3 Purpose of This Assignment

- To help you review templates and other C++ features that you should know for this course.
- To get you into the mindset of implementing a data structure (albeit, a very simple one in this case), since – throughout this course – we will see implementations of different data structures.
- To show you how exactly compile-time index range checking (which `std::array` from the C++ STL supports) can be done. You have seen that sequential STL containers (e.g. `std::vector`) that support indexing do so through either `operator[]` or the `at()` method. The `operator[]` does no index range checking, whereas `at()` does *runtime* index range checking. `std::array` takes it a step further by supporting *compile-time* index range checking, and – by making your own, smaller version of `std::array` in this assignment – you will see how that is done.

4 Reference Environment

The autograder, which is talked about at the end of this document, will compile and run your code in a Linux environment. That means that you should test your code in a sufficiently similar environment. **The CSIF is one such environment.** Each student should have remote access to the CSIF. (I talk more about the CSIF in the syllabus.)

You should avoid causes of undefined behavior in your code (i.e. you should avoid things that make your code behave differently in one environment/context vs. another), such as uninitialized variables. If you have things like this in your code, then your code could end up generating the wrong answer when autograded, even if it generated the right answer when you tested it in an appropriate environment.

5 Programming Problems

Your code for this assignment must be written in C++.

Hopefully, in at least one of your prerequisite courses, the instructor encouraged you to learn how to use a debugger, regardless of the programming language. There are many debuggers that you can use for C++. One that is already on the CSIF (and should be on or easily installable on any other Linux environment) is GDB (`gdb`). GDB is a command-line debugger, and some may not like that, so there is also DDD, a GUI version of GDB. Learning to use a debugger takes some time (just like learning to use the Linux command line), but once you are sufficiently experienced with a debugger, it becomes much easier to find many kinds of bugs. As mentioned in the very first Canvas announcement, I uploaded my lecture slide deck that talks about how to use GDB, just in case it helps.

5.1 File Organization

You will submit two files for this assignment:

- `array.hpp`
- `array.inl`

If you wish, you can write your entire implementation in `array.hpp` and not submit `array.inl`.

6 Templated Array

6.1 Static Assertions

Recall that the `assert()` function[1] can be used to check if a certain condition is true during runtime and crash if the condition is false. We may refer to this as a dynamic assertion.

In contrast, `static_assert()` performs a static assertion[2]. This means that `static_assert()` checks if a condition is true *during compile time* and causes a *compilation error* if the condition is false. Because the check is done during compile time, this allows static assertion conditions to involve values that are known at compile time. This is useful if you are trying to impose certain constraints on template parameters.

Below is an example. Notice that `main2.cpp` cannot be compiled due to the violation of the static assertion.

[1] In C, this is in the `<assert.h>` header. In C++, this is in the `<cassert>` header.

[2] Here, "static" means "without running the program", i.e. during compile-time, whereas "dynamic" means "while running the program". You see these terms "static" and "dynamic" from time to time, e.g. with static profilers vs. dynamic profilers.

```
$ cat test.hpp
template <typename T, int Val>
class Test
{
public:
    // For no reason, let's prevent @val from being 2.
    // (Note: I don't think it matters too much where this static assertion
    // is placed.)
    static_assert(Val != 2, "Val cannot be 2.");

    Test() = default;
};
$ cat main1.cpp
#include "test.hpp"

int main()
{
    Test<float, 5> tmp;
}
$ g++ -Wall -Werror -std=c++14 main1.cpp -Wno-unused-variable
$ cat main2.cpp
#include "test.hpp"

int main()
{
    Test<float, 2> tmp;
}
$ g++ -Wall -Werror -std=c++14 main2.cpp -Wno-unused-variable
In file included from main2.cpp:1:
test.hpp: In instantiation of 'class Test<float, 2>':
main2.cpp:5:20:   required from here
test.hpp:8:23: error: static assertion failed: Val cannot be 2.
    8 |     static_assert(Val != 2, "Val cannot be 2.");
      |                   ~~~~~~~~
$
```

Note: Once the new C++ feature called **concepts** comes out, that would apparently take the place of the combination of templates and static assertions.

6.2 Goal: Implementation of Templated Array

You should start with the `array.hpp` and `array.inl` files that are on Canvas. **In this assignment, your goal is to implement all of the methods that are declared in the definition of the templated `Array` class. There are also non-member functions below the definition of the class that you must define too.**

You *are* submitting both the header file and the source file for this assignment, which means that you *are* allowed to modify the header file.

Below are some tips/suggestions:

- It is possible to avoid using dynamic memory allocation entirely in this assignment.
- Code that involves templates may seem intimidating and more complicated than it actually is. It may help if – in your head – you imagine that `T` is `int` while you are writing the code for the methods.

6.3 Restrictions

You are not allowed to use any STL type, e.g. `std::array`, `std::vector`, as that would totally defeat the point of the `Array` class. You are not allowed to use a linked list as the underlying implementation either; you should be using a C-style array in the underlying implementation. (In this assignment, you are making your own version of the `std::array` class.)

There are certain operations that you can and cannot assume that `T` (the type template parameter given to `Array`) supports. **You may assume the below operations are supported by `T`.** (There might be some obvious ones that I missed, so feel free to ask me if you come across such operations that you think should be in this list.)

- Default constructor.
- Copy constructor.
- Copy assignment operator (`=`).
- `==`
- `!=`
- `operator<<` (That is, you can print out an element of type `T` with, for instance, `std::cout`.)

6.4 Examples

Below are examples of how your code should behave. Do not put `array.hpp` or `array.inl` in the `g++` compilation command.

```
$ cat demo_array.cpp
#include "array.hpp"
#include <iostream>

int main()
{
    std::cout << std::boolalpha;

    Array<int, 3> vals;
    vals.at(0) = 18;
    vals.at(1) = 55;
    vals.at(2) = 49;
    std::cout << "Length: " << vals.size() << '\n';
    std::cout << vals;
    std::cout << "Front: " << vals.front() << '\n';
    std::cout << "Back: " << vals.back() << '\n';

    // This would cause a compiler error because i is a variable and is thus
    // not considered known at compile time.
    // unsigned i = 0;
    // get<i>(vals) = 54;

    Array<int, 2> vals2;
    vals2.at(0) = 35;
    vals2.at(1) = 45;
    std::cout << "Length: " << vals2.size() << '\n';
    std::cout << vals2;

    // This line causes a compiler error, because Array<int, 3> and
    // Array<int, 2> are NOT the same type!!
    // std::cout << (vals == vals2) << '\n';

    Array<int, 2> vals3;
    vals3.at(0) = 35;
    vals3.at(1) = 50;
    // Here, vals2 and vals3 are of the same type, namely Array<int, 2>.
    std::cout << (vals2 == vals3) << '\n';
    std::cout << (vals2 != vals3) << '\n';

    Array<std::string, 3> strs;
    get<0>(strs) = "Hello";
    get<1>(strs) = "there";
    get<2>(strs) = "world";
    std::cout << strs;

    // This line SHOULD cause a compiler error.
    // get<3>(strs) = "blah";

    // This SHOULD cause a compiler error.
    // Array<int, 0> vals4;
    // std::cout << vals4;
}
$ g++ -Wall -Werror -g -std=c++14 demo_array.cpp -o demo_array
$ ./demo_array
Length: 3
18 55 49
Front: 18
Back: 49
Length: 2
35 45
false
true
Hello there world
```

7 Grading Breakdown

As stated in the updated syllabus, this assignment is worth 4% of your final grade. All of this will be from the autograder; there is no manual review for this assignment, except to verify that you obeyed certain explicitly stated requirements.

8 Autograder Details

As stated above, you will submit the below ***two*** files to the autograder on Gradescope. **The names of the files that you submit MUST be correct.** Do not submit any other files.

- `array.hpp`
- `array.inl`

If you wish, you can write your entire implementation in `array.hpp` and not submit `array.inl`.

If your code fails to compile (except when it is supposed to fail), then you will automatically receive a zero.

Your output must match mine *exactly*.

As mentioned in the syllabus, there are hidden test cases on Gradescope whose results you will not be able to see until after the deadline. There are in total 22 test cases: 9 visible and 13 hidden. You can find the inputs used by the visible test cases on Canvas.

Each test case is its own `main()` function. This is to prevent a situation in which if you make an assumption about the operations that `T` – the template type parameter to `class Array` – supports, your code will cause a compilation error that will cause automatic failure of all of the test cases. Thus, it is possible for your code to cause a compilation error on one test case without causing it for all of the others.

In cases in which an exception is expected to be thrown, the autograder does not care if the message/string you use in the thrown exception is correct. In cases in which a compilation error is expected to be caused by a static assertion, the autograder does not care if the message generated by the static assertion is correct.

Two of the hidden test cases check for memory leaks using `valgrind`.

There are a few test cases that check if compilation failed when it was *expected* to fail. Such test cases begin by first confirming that your code can be compiled with other code where compilation is expected to *succeed*. This is to prevent you from passing these test cases automatically simply by submitting code that can't be compiled in the first place.

UCDAVIS
COMPUTER SCIENCE