Assignment #3

CS 225, Summer 2018

Due Date:	9th July 2359hrs
Topics covered:	Type Erasure, Move Semantics, Perfect Forwarding, Template Specialization, Operator Overloading
Deliverables:	To submit one zip file containing the two files `ArrayAdaptor.h` and `SingleArray.h` according to CS225 syllabus naming conventions for the zip file.
Compiler flags :	For gnu, it would be `-std=c++11 -Wall -Wextra -Werror -pedantic` For Visual Studio 2017 (Command line), it would be `/std:c++latest /W4 /WX /nologo /EHa`
Objectives:	To apply the type erasure pattern and other techniques such as perfect forwarding and move semantics.

1 Programming Statement: To design a class template for Array Adaptation

This assignment is motivated by the following realization that in terms of memory layout, the following two variable declarations are similar and yet are accessed by different syntax.

```
int x1[20];
int x2[2][2][5];
```

Both arrays ultimately contain 20 integers. However, `x1` can only be accessed by a single-subscript operator e.g., `x1[5]` while `x2` can be accessed by 3 subscript operator syntax e..g, `x2[0][1][3]`. This assignment is about the design of a class template `ArrayAdaptor` that allows us to access `x1` as if it is an arrays of arrays instead of simply a single array of integer. Furthermore, we will be using type erasure so that we can adapt any random access container e.g, `std::array, std::vector` to be used as if it is an array of arrays instead.

For example, we would like the following to be possible:

```
int bar[20];
ArrayAdaptor<int, 2, 2, 5> adapted_bar(bar);
/* Adapting bar to be an object that can be accessed as if
   it is an array of 2 arrays of 2 arrays of 5 ints i.e., int[2][2][5]*/
std::cout << adapted_bar[0][1][3]==bar[8] << std::endl;
/* always prints 1 */
adapted_bar[1][0][2] = 10; /* modifies bar[12] to 10 */
```

In order to implement this, we would require the implementation of two class templates: `SingleArray<T>` and `ArrayAdaptor<T, NumDims>`. These will be respectively our Task 1 and Task 2.

2 First Task: Type-Erased `SingleArray<T>`

Your first task would be to implement a `SingleArray<T>` class template that supports the following partial interface:

```
template<typename T>
class SingleArray
{
public:
    T operator[](int index) const;
    T& operator[](int index);
};
```

Obviously, this is only part of the public interface we expect from `SingleArray<T>`. The following subsections describe what is needed.

2.1 Type-erasure requirement

Now, we note that `T` here refers to the type that is returned by the subscript operator rather than the type of the object that is used to construct `SingleArray<T>`. For example, we may want to construct `SingleArray<T>` using the following code:

```
SingleArray<int> a1 {new int[10]};
SingleArray<int> a2(std::vector<int>());
```

So both `a1` and `a2` are both `SingleArray<int>` and yet they are providing access to two different types. `a1` gives us access to the `int_array` while `a2` gives us access to the `vec_int`. If we do not see the initialization code, we cannot tell the exact type underlying within each `SingleArray<int>` object, this is known as type-erasure and it's a form of encapsulation.

Key point: `SingleArray<T>` must be constructible with a generic type `U`. In order to access `SingleArray<T>::operator[]` member function properly, `U` must be a type that supports the subscript operator that returns a type `T`. Following from the previous examples, `a1` is constructed with `T=int` and `U=int *` and `a2` is constructed with `T=vec_int` and `U=std::vector<int>`.

2.2 Construction and Assignment requirements

In this section, we detail the requirements for constructing and assigning `SingleArray<T>` objects.

2.2.1 Default construction

First, `SingleArray<T>` objects can be default constructed. Obviously, when `SingleArray<T>` are default constructed (hence no underlying objects), attempt to access it should cause memory error.

```
    SingleArray<int> bar;
    bar[10]; /* should cause memory error */
```

2.2.2 Constructing `SingleArray<T>` using lvalue objects

Second, when a user constructs a `SingleArray<T>` object with a lvalue, the user expects the `SingleArray<T>` object to contain a reference to the object. For example,

```
std::string s {"abcd"};
SingleArray<char> adapted_s {s};
std::cout << adapted_s[3] << std::endl; /* prints 'd' */
adapted_s[2]='f'; /* s is now "abfd" */
s = "hello world";
std::cout << adapted_s[3] << std::endl; /* prints 'l' */
```

2.2.3 Constructing `SingleArray<T>` using rvalue objects

Third, when a user constructs a `SingleArray<T>` object with a rvalue, the user expects the `SingleArray<T>` object to contain a move-constructed copy of the same rvalue-type. For example,

```
std::string s {"abcd"};
SingleArray<char> adapted_s {std::move(s)}; /* Note the std::move here */
std::cout << adapted_s[3] << std::endl; /* prints 'd' */
adapted_s[2]='f'; /* s is NOT affected */
s = "hello world";
std::cout << adapted_s[3] << std::endl; /* prints 'd' */
```

2.2.4 `SingleArray<T>` Copy and Move semantics

Fourth, the copies of `SingleArray<T>`, either acheived through copy construction or copy-assignment, should be assessing the same internal object, whether it was a lvalue reference or a move-constructed rvalue object.

The following code illustrates what is required here:

```
    std::array<short, 10> short_array { 1,2,3,4,5,6,7,8,9,10};
    SingleArray<short> adapted_short_array1 { short_array };
    SingleArray<short> adapted_short_array2 { adapted_short_array1 };
    adapted_short_array1[4]=50;
    /*Now short_array[4] is 50 */

    SingleArray<short> adapted_rvalue_array1 { std::move(short_array) };
    adapted_rvalue_array1[4] = 49;
    /* short_array is unaffected i.e, short_array[4] is still 50*/
    SingleArray<short> adapted_rvalue_array2 { adapted_rvalue_array1 };
    adapted_rvalue_array2[4] = 99;
    /* Now adapted_rvalue_array1[4] is 99 instead of 49*/

    SingleArray<short> adapted_rvalue_array3 = adapted_rvalue_array2;
    adapted_rvalue_array3[9]=8888;
    /* Now adapted_rvalue_array1[9] is 8888] */

    SingleArray<short> adpated_short_array3 = adapted_short_array2;
    short_array[0]=777;
    /* Now short_array[0] is 777 */
```

Furthermore, `SingleArray<T>` is move-constructible and also move-assignable. After move-constructing or move-assigning a `SingleArray<T>` object, the original `SingleArray<T>` is as good as default constructed.

```
    std::array<double, 8> double_array { 1.0, 2.3, 3.5, 4.6,
                                         5.7, 6.8, 7.9, 8.0};
    SingleArray<double> adapted_double_array1(double_array);
    adapted_double_array1[3] = adapted_double_array1[0]
                             +adapted_double_array1[2];
    SingleArray<double>
             adapted_double_array2(std::move(adapted_double_array1));
    /* adapted_double_array2[i] is the same as double_array[i] */
    adapted_double_array1[2]=10.0; /*Segmentation fault*/
    adapted_double_array1 = adapted_double_array2;
    /*Yes. Reassignment should be possible */
```

2.2.5 Summary of first task

The student is expected to design the `SingleArray<T>` class template such that the class template will meet the above requirements. The student should provide their own declaration and definition of the constructors, destructor (if any) and necessary member functions. A driver file named `TestSingleArray.cpp` is provided for the student to test the implementation of the `SingleArray<T>`. All definition and declarations should of `SingleArray<T>` should be given in a file named `SingleArray.h`. This portion of the assignment takes up 50% of the assignment grade.

3 Second Task: Type-Erased `ArrayAdaptor<T, DimSizes...>`

In order to understand the basic capability of the `ArrayAdaptor<T, DimSizes...>`, let us revisit the example we began with.

```
int bar[20];
ArrayAdaptor<int, 2, 2, 5> adapted_bar(bar);
/* Adapting bar to be an object that can be accessed as if
   it is an array of 2 arrays of 2 arrays of 5 ints i.e., int[2][2][5]*/
std::cout << adapted_bar[0][1][3]==bar[8] << std::endl;
/* always prints 1 */
adapted_bar[1][0][2] = 10; /* modifies bar[12] to 10 */
```

How does the expression `adapted_bar[1][0][2]` work? The best way is to consider the observation that `operator[]` is really an operator overloading, which is a member function.

Now, `adapted_bar` is an object of type `ArrayAdaptor<int, 2, 2, 5>`. What would be the return type when we call it's `[]` operator? Well, I suggest that it should return an `ArrayAdaptor<int, 2, 5>` object.

What type should we return when we call the `[]` operator of the `ArrayAdaptor<int, 2, 5>` object? We should return an `ArrayAdaptor<int, 5>` object.

Finally, when we call the `[]` operator of the `ArrayAdaptor<int,5>` object, we should return a `int` object.

Therefore, a **partial** declaration of the `ArrayAdaptor<T, DimSizes...>` class look like this:

```
1
2  template< typename T, unsigned FirstDimSize,  unsigned ... RestDimSizes>
3  class ArrayAdaptor
4  {
5  public:
6      using SubscriptResultType = ArrayAdaptors<T, RestDimSizes...>;
7      SubscriptResultType operator[](int index);
8      SubscriptResultType operator[](int index) const;
9      ...
10 };
11
12 template< typename T, unsigned OnlyOneDim>
13 class ArrayAdaptor<T, OnlyOneDim>
14 {
15 public:
16     T& operator[](int index);
17     T operator[](int index) const;
18     ...
19 };
```

If you have followed the description so far, then these are what is left for the students to figure out:

- Firstly, there must be some way to keep track of the indices that is accessed so far. For example, given an array `int x[100]`, if we access it as if it is a "3-dimensional" array `int adapted_x[10][2][5]`[1]. How should we compute the desired index accessed if say, we are accessing `adapted_x[i][j][k]`? The computation would be `i * 10 + j * 5 + k`. But the problem is that the reality is that each `[]` operator call is performed on a different object, so there must be way to preserve the information of what the previous indices have accessed.

- Secondly, note that we are type-erasing the underlying type right from the start of `ArrayAdaptor`. Students should take note that this does not mean that we will immediately apply type-erasure pattern to ArrayAdaptor (in fact, those who try to do so may find themselves stuck at certain points.) Rather, consider how you can make `SingleArray<T>` a sub-object of `ArrayAdaptor` that can be passed on through the diminishing dimensions until we need to return `T` instead of some `ArrayAdaptor` type.

[1]The student should realize that C++ doesn't support multi-dimensional arrays. Rather, static arrays such as `int y[10][5][8]` should properly be called an array of 10 arrays of 5 arrays of 8 integers. What these really are in terms of their types would be arrays of arrays. But for the sake of reducing verbosity, we refer to such constructs as multi-dimensional arrays.

The student is expected to design the `ArrayAdaptor<T, DimSizes...>` class template such that the class template will meet the above requirements. We would not test the Copy control and move semantics of the `ArrayAdaptor`, though the student should take note that his/her design will not compromise the performance of the program too much. The student should provide their own declaration and definition of the constructors, destructor (if any) and necessary member functions. A driver file named `TestArrayAdaptor.cpp` is provided for the student to test the implementation of the `ArrayAdaptor<T, DimSizes...>`. All definition and declarations should of `ArrayAdaptor<T, DimSizes...>` should be given in a file named `ArrayAdaptor.h`. This portion of the assignment takes up 50% of the assignment grade.

3.1 Additional Requirement

The student is expected to complete the above two tasks **without writing loops both explicit and implicit (such as calling `for_each`)**. Those who violate this constraint will lose 40% of the entire grade for this assignment.