Assignment #5

CS 225, Summer 2018

Due Date:	15th August 2018 2359hrs
Topics covered:	Class template, function template, template specialization, Meta-programming, pointer-to-members, variadic templates
Deliverables:	To submit one zip file containing the one file `gensoa.h`
Compiler flags :	For gnu/clang, it would be `-std=c++1z -Wall -Wextra -Werror -pedantic` ~~For Visual Studio 2017 (Command line), it would be `/W4 /WX /std:c++latest /nologo /EHa`~~ We're not testing this for Visual Studio for this assignment
Objectives:	To practise the various techniques when implementing templates.

Programming Statement: To provide a facility that generates and converts arrays of structs into structs of arrays

We begin by consider the following the motivating example:

```
struct Point
{
    float x, y, z;
};
std::array<Point, 5> array_of_points; //array of 5 Point objects

template<size_t NumOfElements>
struct Points
{
    std::array<float, 5> xs, ys, zs;
};
```

A `Points<5>` object would contain the same number of floats as the `array_of_points` – 15 floats. However, the memory layout for both would be different. For certain programs where the `xs` of all the objects are accessed before the `ys` and so on, the latter struct would provide a better performance in terms of cache behaviour. [1]

In this assignment, we would like to explore the possibility of having a scalable solution to how we may generate the struct of arrays (the latter object structure) from a given array of structs. Ideally, we would like the following to be possible:

```
struct Haha {int x; float y; std::string z; /*Some Reflection Code*/};
std::array<Haha, 5> array_of_hahas;
auto HahaSOA = GetSOA(array_of_hahas);
//HahaSOA now is a struct of an array of 5 ints, 5 floats and 5 strings.
```

The rest of this document describes the task necessary for the student to accomplish for this.

[1] The proper terms would be that better **spatial locality** is being exploited. We invite the reader to find out more regarding that on his/her own as that's not the main point of the programming assignment.

`StructOfArray` class template

Now, automatically generating a struct of arrays type with "reasonable names" - e.g., converting `x` into `xs` in the initial example is nigh impossible, apart from usage of certain kinds of C++ macros magic. However, we constrain ourselves in this assignment to what is possible within the standards of C++.

We would **NOT** be testing the `StructOfArray` class directly, especially in terms of instantiating it with specific template arguments. However, the following is what can be conceived as a reasonable partial declaration of the `StructOfArray`. At the same time, the student must ensure that `StructOfArray` will ensure that the memory layout has been rearranged as stated in the beginning of the specification.

```
template<unsigned NumElements, typename ... Types>
struct StructOfArray
{
    void print(std::ostream& ) const;
private:
    std::tuple< std::array<MemTypes, NumElements>... > arrays;
};
```

Instead of giving each `array` a specific name within the `StructOfArray` we use a `tuple` to store the various arrays. The `tuple` is a particular useful container for these arrays because `tuple` can store heterogeneous types.

To make this concrete, we describe the actual type of `HahaSOA` in the running example above:

```
struct Haha {int x; float y; std::string z; /*Some Reflection Code*/};
std::array<Haha, 5> array_of_hahas;
auto HahaSOA = GetSOA(array_of_hahas);
// The type of HahaSOA would be - instantiated StructOfArray<5, int, float, ↩
    std::string>
/*
struct StructOfArray<5, int, float, std::string>
{
private:
    std::tuple< std::array<int, 5>,
                std::array<float, 5>,
                std::array<std::string, 5>
              > arrays;
};
*/
```

Due to some of the challenges with storing array types and reference types, we assume that the `Types` parameter pack of the `StructOfArray` can only contain types with the following properties (or rather, we will only :

- Copyable
- Not C++ native array (i.e., `std:array` is possible, but see the 3rd point below)
- has an overloaded output stream operator `<<`

The `StructOfArray` is expected to hold copies of the fields (it's supposed to perform like a separate data structure instead of holding references to the original struct/class.

However, while we give the students a freedom in designing the actual declaration and implementation of the `StructOfArray` (in fact, you may choose to rename it as you like, but please be mindful of code quality and readability), the driver test cases expect each of the `StructOfArray` objects to support a `print` member function. The declaration of the function is the following:

```
void print(std::ostream& ) const;
```

This function will be printing the elements of each array in a row,and the arrays in successive rows. The following shows the expected behaviour of the `print` function.

```
struct Haha {int x; float y; std::string z; /*Some Reflection Code*/};
std::array<Haha, 2> array_of_hahas { Haha{3, 6.2f, "abc"}, Haha{44, 7.55f, "whatever"}};
auto HahaSOA = GetSOA(array_of_hahas);
//HahaSOA now is a struct of an array of 2 ints, 2 floats and 2 strings.
HahaSOA.print();
/* prints:
3 44
6.2 7.55
abc whatever
*/
```

The `GetSOA` function template

As indicated above, the driver tests will not be instantiating the `StructOfArray` objects directly. Rather, all such objects will only be instantiated through using the `GetSOA` function template, which has the following signature:

```

template<size_t NumOfElements, typename C>
auto GetSOA(const std::array<C, NumOfElements>& array_of_Cs);
```

The `GetSOA` function is supposed to do the following:

- Figure out the Pointer to Member types for class `C`.
- Figure out the exact `StructOfArray` type
- Construct the `StructOfArray` type using the given `std::array` of `C` objects.
- Return the constructed `StructOfArray` object.

The first two objectives of the `GetSOA` are the most challenging part of this assignment. We now describe what may be of help with the following hints:

1. `GetSOA` will only be successfully compiled with reflected User Defined Types. We provide a `reflection.h` file, which is similar to what is shared in class, with the `MetaData` object having an additional member function - `GetPtrToMembers`, which has the following function prototype:

```
const auto& GetPtrToMembers() const
```

What it returns is a `tuple` of all the pointer to members that are registered with the `MetaData` object.

2. But since the `MetaData` only provide pointer to members, we need to perform some tricks in order to obtain the types of the variables without dereferencing any objects of type `C`. [2] This requires careful use of the `decltype` and `declval` supported by C++11/14 onwards.

3. `decltype(expr)` and given that the `expr` is of type `T` yields the following:

   - T&& if `expr` is an xvalue expression.
   - T& if `expr` is an lvalue expression.
   - T if `expr` is prvalue expression.

```
decltype(1) x; // x is an int.
decltype(x) y; // y is an int. Names of variables are treated as ↩
    prvalue. (Yes I know, it's confusing.)
decltype((x)) z ; //NC. z is an int&
decltype((x)) z = x; //OK. binds int& z to x.
decltype(std::move(x)) a = x; // NC. a is int &&
decltype(std::move(x)) b = 5; // OK. a is int &&
```

4. `declval<T>()` is a function template that's only declared (not defined) and it can be used in conjunction with `declval` to find types of fields without constructing `T`.

```
namespace std
{
template<class T>
typename std::add_rvalue_reference<T>::type declval() noexcept;
//declaration of declval in std, found in <utility>
}
// Usage
struct Hehe { int & a;  float b }; //Hehe cannot be default constructed↩
    .
int i; float f;
decltype(std::declval<Hehe>().a) x; //NC. x is a int &.
decltype(std::declval<Hehe>().a) x = i; //OK.
decltype(std::declval<Hehe>().b) y = f;  //OK. y is a float&.
```

[2] Some students may object to this because the `GetSOA` function takes in a array of `C` objects. However, please note that `std::array<int, 0>` is technically compile-able, but not accessible. So we do need to avoid dereferencing real `C` objects when finding out the pointee types.

Your Task

The student is expected to be able to provide the declaration and definition, together with any auxiliary functions/classes in a file called `gensoa.h`. In terms of rubrics, apart from the usual (tablen, indentation check, doxygen-style comments, code quality check, resource leak etc), the students are expected to:

- Pass all the provided test cases. (90%).
- Pass the hidden test cases (10%).