Assignment 6.

Type erasure pattern and allocator classes

Purpose of the exercise

This exercise will help you do the following:

- Demonstrate understanding of templates and the type erasure pattern (TEP).
- Develop familiarity with allocator classes.
- Practice use of smart pointers.

Requirements

Previously, you were expected to implement a class template similar to `std::bitset<N>`. Any function that accepts a parameter of such type has to either accept a parameter of a class type with a specific `N`, or be a function template with a non-type template parameter, which it could use as `N` within the template. Alternatively, the code could abstract away the actual parameter type using the **type erasure pattern**.

This is exactly the case for the functions in the provided test driver *Test_Bitset.cpp*:

```
typedef cs225::bitset_tep tep;

void test1_impl(tep bitset, size_t index);
//...
void test6_impl(tep bitset);
```

In sample declarations above, `tep` is a non-templated alias to the main class you must implement. Your task is to implement a type erasure pattern (TEP) class to encapsulate a `bitset<N>` instance of any size `N`. Your type should expose a static member function template `create<T>()` that given any type `T` with an implicit interface (*duck-typing*) of a `bitset<N>` produces a type-erased object. Additionally, there are a few more implementation features that you have include in this assignment.

All your tasks have been summarized below:

- Implement a simple class `allocator` (used by *Test_Bitset.cpp*, line 21).

  Objects of this class allocate and deallocate arrays of elements as requested. Your implementation must perform appropriate printouts with each operation; see *expected-output.txt* for details.

  - Provide the class definition in *allocator.h*; define member functions in *allocator.cpp*.
  - Use *raw pointers* as needed.
  - Review a related documentation of a class template std::allocator available in the Standard Template Library before implementing your code.
- Update your implementation of `bitset<N>` to `bitset<size_t N, typename Allocator>`.

  Delegate a responsibility (*Single Responsibility Principle*) of memory management to an allocator class. Do this by adding a `typename` parameter to the class template of a bit set, then add a data member of this type inside the class. Instead of allocating or deallocating memory directly, call functions `allocate(n)` and `deallocate(ptr, n)` of that member.

    - Keep the class template definition in *bitset.h*; define member functions in *bitset.hpp*.
    - Do not use `new`/`delete`, `malloc`/`free`, or related functions in this class template; use an allocator data member instead.
    - Use *raw pointers* as needed.
- Implement `bitset_tep` class, not a template!

  TEP should expose all the functionality required for it to work with the test cases as a bit set. The test cases are the same as in the previous assignment (except that there is no test cases assessing the size of the object), but now they work on the TEP object, not a bit set of a specific size.

    - Provide the definition of classes inside *bitset_tep.h*; define member functions in *bitset_tep.cpp* and template's member functions inside *bitset_tep.hpp*.
    - Remember to properly define an interface class `IConcept` and derive from it a class template `Model<T>`; both these definitions should be dependent types of a class `bitset_tep`.
    - Define only the functionality needed by the test driver.
    - Entire implementation of TEP must **not** use any raw pointers, nor the operators:
        - `operator*` (dereference),
        - `operator&` (address-of),
        - `operator new` (allocation) in any version, `malloc()` or any similar function,
        - `operator delete` (deallocation) in any version, `free()` or any similar function.
    - Pay attention to proper encapsulation, `const`-correctness and good practices.
    - Observe the Rule of 3/5/0.

To check the behavior of the test cases with your implementation of TEP (without using `allocator`), you can compile the code with the `USE_STL_BITSET` pre-processor symbol.

To detect for memory leaks, define a symbol `MEMORY_DEBUG` used by *MemoryAlloc.h*:

```
g++ -DMEMORY_DEBUG -Wall -Werror -Wextra -Wconversion -pedantic -std=c++17 -o
main test_bitset.cpp allocator.cpp bitset_tep.cpp
```

Requested files

Without any comments you can expect files to be around the following sizes:

- *bitset.h* - same as your original implementation.
- *bitset.hpp* - same as your original implementation.
- *allocator.h* - 15 lines.
- *allocator.cpp* - 15 lines.
- *bitset_tep.h* - 100 lines.
- *bitset_tep.cpp* - 50 lines.
- *bitset_tep.hpp* - 25 lines.

No *checklist* or comments are required; it is a good practice to include them, but the automated grading tests will not penalize you for the lack of comments, at least in this assignment.

At this level of the course, it is expected that the style of your code is elegant, easy to read and has a desired quality of being self-explanatory and easy to understand. The automated grading tests will not penalize you for low quality of the code working properly, at least in this assignment.

Submitting the deliverables

You have to upload requested files to Moodle - DigiPen (Singapore) online learning management system, where they will be automatically evaluated.

To submit your solution, open your preferred web browser and navigate to the Moodle course page (pay attention to the section name suffix at the end of the course name). In the course page find a link to the Virtual Programming Lab activity that you are submitting.

In the **Description** tab of the activity you can find a due date, and a list of requested files. When you switch to the **Submission** tab you should see the controls for uploading or typing exactly the files that are required. Upon clicking the *Submit* button the page will validate submitted files and report any errors. If the submission was successful, the page will display a message that the submission has been *Saved*. You should press the *Continue* button to see the *Submission view* page with the results of the evaluation and the grade.

If you received an *A* grade, congratulations! If not, before the due date you can still review and update your solution and resubmit again. Apart from exceptional circumstances, all grades after the due date are final, and students who did not submit their work will receive a grade *F*.