Assignment 2.

Inheritance and interface classes

Purpose of the exercise

This exercise will help you do the following:

1. Refresh your knowledge about object-oriented programming, in particular class inheritance.
2. Develop familiarity with interface classes.
3. Highlight the importance of using virtual destructors in base classes.

Requirements

Suppose that you are writing a program for saving formatted text into text files and want to support various file formats. In this assignment you will support:

- Plain-text files.
- Markdown
- HTML5

In such case, your program would typically not rely on a particular implementation of a class saving the content, but rather define an **interface class** that represents required capabilities of each such class; these classes must inherit from the interface class to provide implementation *realizing* that interface.

In this assignment, you receive a driver code that constructs a formatter object and uses it to save formatted text. The driver uses an interface class named `IFormatter`. This interface class, to keep things simple, supports 4 types of elements:

- Level 1 Headings,
- Level 2 Headings,
- Paragraphs of text,
- Block quotes.

For how they are represented, study the expected output files.

Your task is to define the interface class `IFormatter` inside *IFormatter.h*, and provide three implementations classes through the files listed below:

- `Text` (*Text.c*, *Text.h*).
- `Markdown` (*Markdown.c*, *Markdown.h*).
- `Html` (*Html.c*, *Html.h*).

Each translation unit will be compiled separately and individually linked with the object file of the driver. Then it will be used to generate the actual output that for each of 3 files will be compared with the expected output. The output must match exactly.

While implementing the files you must observe the following constraints:

- *IFormatter.h* may include only `<string>` header (optional).
- All other source files may only include header files included in this assignment; no standard headers are allowed.
- You are not expected to support functionality and formatting not directly required in this assignment neither related to Markdown nor HTML5 formats.

- All formatting member functions of `IFormatter` are expected to return `IFormatter&` so that the driver code can successfully *chain* calls to all member functions in a single expression.
- No function definitions are allowed in header files.

To compile individual translation units, link them into executables, and generate output you can run the following commands:

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

g++ -DMEMORY_DEBUG -Wall -Werror -Wextra -Wconversion -pedantic -std=c++17 -c -o text.o text.cpp
g++ main.o text.o -o main
./main > output.txt 2> mem.log

g++ -DMEMORY_DEBUG -Wall -Werror -Wextra -Wconversion -pedantic -std=c++17 -c -o markdown.o markdown.cpp
g++ main.o markdown.o -o main
./main > output.md 2>> mem.log

g++ -DMEMORY_DEBUG -Wall -Werror -Wextra -Wconversion -pedantic -std=c++17 -c -o html.o html.cpp
g++ main.o html.o -o main
./main > output.html 2>> mem.log

cat output.txt output.md output.html mem.log > output
```

A preprocessor macro `MEMORY_DEBUG` allows *MemoryAlloc.h* file to perform checking whether individual programs display any free store memory leaks.

There will be 50 test cases corresponding to 50 lines of the *output* file as shown in the script above:

- 1 per line of output provided by a `Text` formatter (11 total)
- 1 per line of output provided by a `Markdown` formatter (13 total).
- 1 per line of output provided by an `Html` formatter (19 total).
- 1 per line of output related to memory leaks (3 programs 2 lines each; 6 total).
- 1 empty line at the end.

As each of the individual output files is a plain text file, it can be inspected in a plain-text editor.

Requested files

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

- *IFormatter.h* - 15 lines.
- *Text.c* and *Text.h* combined - ~70 lines.
- *Markdown.c* and *Markdown.h* combined - ~85 lines.
- *Html.c* and *Html.h* combined - ~100 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

Every assignment specification, like this one, will instruct you what deliverables are expected as a result of the assignment. You have to upload these 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 can 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*.