Assignment 3.

Virtual function calls and the dependency inversion principle

Purpose of the exercise

This exercise will help you do the following:

1. Refresh your knowledge about object-oriented programming, in particular dynamic polymorphism.
2. Practice object-oriented design based on interface classes.
3. Practice implementation of virtual function calls.
4. Practice the dependency inversion principle.

Requirements

In the **Assignment 2.** you wrote a program for saving formatted text into text files with support for:

- Plain-text files.
- Markdown
- HTML5

Each of these formatter implementations resided in a separate translation unit and had to be statically linked with the main program's translation unit to produce an executable using that formatter.

In this assignment the program will support all these formatters at the same time, and let the user choose during execution the one to be used. Through this program you will practice two function virtualization techniques:

- Virtual construction (with the factory method design pattern), and
- Virtualization of non-member functions (via delegation to a virtual member function)

These two mechanisms will allow the program to follow **the dependency inversion principle**: the *main.cpp* translation unit should be compileable without any formatter implementation included! All implementations should be included in *Constructor.cpp* translation unit only and instantiable through functions declared in *Constructor.h* (and defined by you in *Constructor.cpp*).

In other words, *main.cpp* should not know about any particular formatter. During compilation it is the formatters that depend on its code (specifically, on interfaces defined for use by *main.cpp*). However, during runtime the `main()` function indirectly instantiates a formatter and calls its function, making the flow of control dependent on a formatter's implementation. The code from *main.cpp* does not know about any formatters, but it uses them; this is different from a more simplistic approach where the code in the `main()` function knows about formatter types and both instantiates them directly and calls their functions. Allowing for such design and following the *dependency inversion principle* let's you write the code that it loosely coupled and easier to reuse or extend: adding a new formatter should not require recompilation of *main.cpp*, just linking with a recompiled *Constructor.cpp*.

The function `main()` should not instantiate a formatter directly; it should do so through a virtual construction based on a choice made by a user during execution of the program. One of formatter factory classes will be instantiated to create headers, paragraphs and other elements of a document in a format-specific fashion. This implementation will represent the **factory method design pattern** in a version that can perform virtual construction without any parameters provided. For example the following code will result in instantiation of a different element implementation for each formatter:

```
IFormatterElement* element = factory->create_begin();
```

This is because the `factory` pointer will point to an address of a derived factory object of a formatter-specific type.

A similar approach is applied in a function `save()`. This global function, which must be implemented by you, is called from the function `process()` for any element realizing an interface `ISavable` to save it into a given stream. The challenge here is to ensure that each element is saved into a stream with its specific formatting; it will be a polymorphic call from a global function, which itself cannot be `virtual`. An interface `ISavable` represents a capability of a class instances to save their contents to a file - if an element does not need to be saved to a file, make sure its class does not realize this interface class. For example, in plain-text files the beginning of the document does not require any text - such element should not realized `ISavable` interface.

Your task consists of 2 parts. In the first part complete and compile (without linking) *main.cpp*:

- Define an interface class `IFormatterFactory` (*IFormatterFactory.h*) that supports all operations required by `main()`.
- Define an interface class `IFormatterElement` (*IFormatterElement.h*) that supports all operations required by `process()`.

When you successfully completed this part, you should be able to compile *main.cpp* like so:

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

As you can see, *main.cpp* should be compileable without any formatters. Implementing them is the second part of your task:

- Define formatter classes that realize `IFormatterFactory`; in the scope of each class you can include other dependent classes that are needed for its implementation. Respective header files should include definitions of classes and definitions of all member functions as well.
  - `Text` in *Text.h*
  - `Markdown` in *Markdown.h*
  - `Html` in *Html.h*
- Define functions from *Constructor.h* and used by *main.cpp* inside *Constructor.cpp*:
  - `get_options()` for retrieving a collection of formatter names,
  - `construct()` for instantiating a factory for a selected formatter,
  - `save()` for saving objects that realize `ISavable` to a stream.

When you successfully completed this part, you should be able to compile *Constructor.cpp* like so:

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

With both translation units compiled, you can link them into an executable and test:

```
g++ main.o constructor.o -o main
echo 1 | main > output.txt  2>  mem.log
echo 2 | main > output.md   2>> mem.log
echo 3 | main > output.html 2>> mem.log
```

While implementing the files you must observe the following constraints:

- Permitted headers:
  - *IFormatterElement.h* must not include any files.
  - *IFormatterFactory.h* may only include *IFormatterElement.h* and <*string*>.
  - *Text.h*, *Markdown.h*, *Html.h* may only include *IFormatterFactory.h* and *ISavable.h*.
  - *Constructor.cpp* may only include *Constructor.h*, *Text.h*, *Markdown.h*, *Html.h*.
- You are not expected to support functionality and formatting not directly required in this assignment neither related to Markdown nor HTML5 formats.
- This time function definitions are allowed in header files.

To compile individual translation units, link them into executables, and generate output you can run the following Linux commands. In Microsoft Windows remove `./` and replace the command `cat` with `type`.

```
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 constructor.o constructor.cpp

g++ main.o constructor.o -o main

echo 1 | ./main > output.txt  2>  mem.log
echo 2 | ./main > output.md   2>> mem.log
echo 3 | ./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 71 test cases corresponding to 71 lines of the *output* file as shown 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 9 lines each; 27 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:

- *IFormatterElement.h* - 10 lines.
- *IFormatterFactory.h* - 25 lines.
- *Constructor.cpp* - 35 lines.

- *Text.h* combined - 55 lines.
- *Markdown.h* combined - 65 lines.
- *Html.h* combined - 70 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 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*.