Programming Assignment: Object-Oriented Battleship

Learning Outcomes

- Gain experience in data abstraction and encapsulation concepts in C++: classes, objects, constructors, destructor

Task

This assignment will give you some practice with object-oriented programming: classes, objects, constructors, and destructors. The task is to refactor the battleship assignment previously implemented as a procedural program in Programming Assignment 3 [PA3] into one that uses C++ classes to implement data abstraction and data encapsulation. You should be able to re-use almost all of your code from the previous assignment. You'll have to go through the following steps to complete the assignment:

1. Make a copy of your PA3 project. Review the handout from PA3 to recall the functionality that was implemented.
2. Replace driver source ocean-driver.cpp with a new driver source file ocean-driver2.cpp [download from assignment web page].
3. Remove all output files and download the test?.txt files from the assignment web page.
4. You'll now refactor ocean.h and ocean.cpp to replace the current `struct Ocean` and related interface functions with an object-oriented version:
    1. Replace `struct Ocean` with `class Ocean` with data members encapsulated using a `private` access specifier.
    2. Constructor for class `Ocean`: You've previously declared initialization function `CreateOcean` in ocean.h and defined it in ocean.cpp. The declaration should be renamed as a constructor and moved into the `public` section of class `Ocean`. The `CreateOcean` definition in ocean.cpp must be refactored into a constructor definition. You must watch out for a nasty surprise that arises when a parameter name in a constructor or member function is the same as a data member. Consider the following code fragment where constructor parameters have the same names as class data members:

```
class NamedInt {
public:
  NamedInt(std::string const& name, int value) {
    name = name;   // self-assignment of left parameter
    value = value; // self-assignment of right parameter
  }
  // other stuff ...
private:
  std::string name;
  int value;
};
```

In constructor `NamedInt`, parameters `name` and `value` are *hiding* the data members from the compiler. Therefore, the assignments on lines 4 and 5 are self-assignments of the parameters.

On the other hand, a constructor member-initialization list will make data member names' visible to the compiler:

```
1  class NamedInt {
2  public:
3    NamedInt(std::string const& name, int value)
4      : name{name}, value{value} {} // data members are
5                                    // properly initialized
6    // other stuff ...
7  private:
8    std::string name;
9    int value;
10 };
```

> ***It is never a good idea to define member functions with parameters having same names as data members.***

3. Destructor for class `Ocean`: Similarly, you must refactor the declaration and definition of `DestroyOcean` into a destructor for `Ocean`.

4. Remaining interface for class `Ocean`: The current interface functions `GetShotStats`, `TakeShot`, and `PlaceBoat` must be refactored into public member functions for `Ocean`. Studying driver source file ocean-driver2.cpp, you'll realize that `GetDimensions` and `GetGrid` are two additional member functions that must be defined as part of `Ocean`'s interface.

5. Using `Ocean`'s interface, it is now possible for clients to implement their own functions [such as function `Dump` in ocean-driver2.cpp] to print debug information for `Ocean` objects. Therefore, you should delete the declaration of function `DumpOcean` in ocean.h and the corresponding definition in ocean.cpp. However, the definition of function `Dump` in ocean-driver2.cpp references `const` variable `HIT_OFFSET` defined in ocean.cpp. Unlike C, C++ global constants have static linkage. If you try to use a global constant in C++ in multiple files you'll get an unresolved external error. The compiler optimizes global constants out, leaving no space reserved for the variable. One way to resolve this error is to include the `const` initializations in a header file and include that header in your C++ source files when necessary, just as if it was a function declaration. The C++17 way of doing this is to move the definitions of `HIT_OFFSET` and `BOAT_LENGTH` from ocean.cpp into ocean.h like this:

```
1  namespace HLP2 {
2    namespace WarBoats {
3      inline int const BOAT_LENGTH {4};   //!< Length of a boat
4      inline int const HIT_OFFSET  {100}; //!< Add this to the boat ID
5
6      // other stuff ...
```

Using keyword `inline` [new in C++17], you're telling the compiler not to define these two `int` objects in every file that includes ocean.h, but rather to collaborate with the linker in order to place it in only one of the generated object files. Note that this use of `inline` has nothing to do with the use of `inline` for avoiding function calls by executing the function body at the call site.

6. That's all. If your implementation was correct in PA3, you should be able to compile, link, and execute with similar results.
7. Not necessary but something to try: You can prevent default construction and copies by `delete`ing the default constructor, copy constructor, and copy assignment operator.

Submission Details

Please read the following details carefully and adhere to all requirements to avoid unnecessary deductions.

Submission files

You will submit ocean.h containing a definition of class `Ocean` and ocean.cpp containing the implementation details of class `Ocean`.

Compiling, linking, and testing

The correct output files for unit tests executed by the driver are labeled test?.txt. Testing your output and checking for memory leaks and errors is doable by explicitly typing out commands in the Linux shell for one or two tests. However, it can become overly cumbersome when having to repeat for more than a couple of tests. This is where an automation tool like *make* shines.

Run *make* with the prep rule to create the release and debug directories in the current working directory:

```
$ make prep
```

Run *make* with the release rule to create the non-debug program executable ocean.out up to date in ./release:

```
$ make release
```

There are six tests specified by command-line parameters 1, 2, and so forth. To run test 1 using *make*, you'd run the command:

```
$ make test1
```

The output file your-test1.txt is created in the ./release directory and the diff command is given this file and my [correct file] test1.txt. If the diff command is not silent, your output is incorrect and your code must be debugged.

To run all tests, run the command:

```
$ make test-all
```

To ensure there are no memory leaks or errors, you must use Valgrind to analyze the runtime behavior of your program in relation to the memory allocated from the free store. Since Valgrind requires option -g, you need a different target. The makefile contains a target called debug-memory. Run the *make* command with target debug-memory like this to create the debug version of the executable in directory ./debug:

```
$ make debug-memory
```

To run test 1 with Valgrind, you'd run the command:

```
$ make debug-test1
```

In this case, the output file your-debug-test1.txt is created in the ./debug directory and the diff command is given this file and my [correct file] test1.txt. If the diff command is not silent, your output is incorrect and your code must be debugged.

To run all tests, run the command:

```
$ make debug-test-all
```

If you want to get rid of all object, executable, and output text files, run *make* with the target clean:

```
$ make clean
```

File-level and function-level documentation

Every source and header file *must* begin with a *file-level* documentation block. This module will use Doxygen to tag source and header files for generating html-based documentation. In addition, every function that you declare and define and submit for assessment must contain *function-level documentation*. This documentation should consist of a description of the function, the inputs, and return value.

Submission and automatic evaluation

1. In the course web page, click on the appropriate submission page to submit the necessary files.
2. Please read the following rubrics to maximize your grade. Your submission will receive:
   - F grade if your submission doesn't compile with the full suite of g++ options.
   - F grade if your submission doesn't link to create an executable.
   - Your implementation's output must exactly match correct output of the grader [you can see the inputs and outputs of the auto grader's tests]. There are only two grades possible: $A+$ grade if your output matches correct output of auto grader; otherwise F.
   - A deduction of one letter grade for each missing documentation block in your submissions. Each source file must have **one** file-level documentation block and a function-level documentation block for each defined function. A teaching assistant will physically read submitted source files to ensure that these documentation blocks are authored correctly. Each missing or incomplete or copy-pasted [with irrelevant information from some previous assessment] block will result in a deduction of a letter

grade. For example, if the automatic grader gave your submission an $A+$ grade and one documentation block is missing, your grade will be later reduced from $A+$ to $B+$. Another example: if the automatic grade gave your submission a C grade and the two documentation blocks are missing, your grade will be later reduced from C to F.