Week 6 — *Homework: C++ classes*

The goal of the present exercise is to program a family of objects to compute series and to dump them. The following points will be considered for the grading:

- Proper usage of git (meaningful comments, several commits with developments steps, use of .gitignore)
- Code works as intended
- Readability of the code (meaningful variable names, comments)
- Minimal documentation: README file

Exercise 1: *Creating a Project directory*

A good coding practice is to use a structure for files. Through this exercise, you are asked to create the following directory structure:

```
homework2/
-- src/    # contains .hh and .cc files
   -- CMakeLists.txt
   -- main.cc
-- build/  # folder for compilation (not to be commited)
CMakeLists.txt
.gitignore
README.md
```

1. Create an empty `main.cc` in `src` folder. Create a `CMakeLists.txt` to compile the `main.cc` in a separate `build` folder. Run the generated executable. Remember for the above directory structure there will be two `CMakeLists.txt` files. One at root level and other inside the `src` folder. The `CMakeLists.txt` at root level will have the following structure:

   ```
   cmake_minimum_required(...)
   add_subdirectory(...) # directory you want to add or contains executable files
   ```

   The `CMakeLists.txt` in `src` folder will add the executable files.

   ```
   add_executable(...)  # .cc files
   ```

2. Create a `.gitignore` which contains necessary file/file extensions that are not to be pushed to the repository.
3. Git push the current state of the project as your first commit.

We will maintain the above directory structure for the rest of the exercise.

Exercise 2: *Series class*

In this exercise, we encourage you to use algorithms from the Standard Template Library (STL) as much as possible. Also, please remember that class definitions and method declarations should be put in .hh files whereas method implementations should be put in .cc files.

1. What is the best way/strategy to divide the work among yourselves? Please answer this question in the **README** file.

2. Create a class named **Series** with the pure virtual function

```
class Series{

 public:

 virtual double compute(unsigned int N) = 0;
}
```

This simple class will represent the interface of the family of classes inheriting from **Series**.

3. Create a class named **ComputeArithmetic** which inherits from **Series** and which implements the series:

$$S_n = \sum_{k=1}^{N} k$$

in the function

```
double compute(unsigned int N);
```

4. Create (instantiate) a **ComputeArithmetic** object in the **main** and call the **compute** method, to output the result to screen (using *std::cout*), and test the program. You may need to *#include* the input-output stream objects of the standard library. Look for the proper header on the web.

5. Create another **ComputePi** class which computes the series converging towards *pi*:

$$\pi = \sqrt{6 \sum_{k=1}^{N} \frac{1}{k^2}}$$

6. Modify the main to decide at execution time which of **ComputeArithmetic** or **ComputePi** is to be instantiated and store the result in a **Series** pointer. Be carefull regarding the memory allocation when manipulating pointers.

Exercise 3: *Dumper class*

1. Create a class named **DumperSeries** of the kind

```
class DumperSeries{

  public:

  virtual void dump() = 0;

  protected:

  Series & series;
}
```

Please note the pure virtual function which says that this simple class will represent the interface of the classes inheriting from **DumperSeries**.

2. Create a class named **PrintSeries** which inherits from **DumperSeries**. This class will output (to screen) every step out of frequency that is lower than *maxiter*. Thus frequency and *maxiter* should be declared as members and set by the constructor. Implement the method **dump**.

3. Create (instanciate) a **PrintSeries** object in the **main** and call the **dump** method to output the previously created **Series** reference.

4. In the **Series** class, append the method

```
virtual double getAnalyticPrediction();
```

which provides the analytic prediction of a class. The default should return **Not a Number** (use the function *nan()*). Implement the correct method in the **ComputePi** class.

5. In the **dump** function of the **PrintSeries** class, use the analytic prediction to provide the convergence towards the limit (only when the analytic prediction is available, i.e. is not NaN).
6. Create another class named **WriteSeries** which also inherits from **DumperSeries**. Implement the method **dump** which writes numerical results of all steps lower than *maxiter* in a file. It should also write the analytical prediction.
7. Implement a function **setSeparator** in **WriteSeries** which takes separator/delimiter as an argument. Depending on the separator, the method **dump** should generate different file formats. For example, comma (,) separator should generate a `.csv` file, space/tab ( ) separator should generator a `.txt` file and pipe (|) separator should generate `.psv` file. By default, it should write a `.txt` file.
8. Create a python script which reads the file and plots the numerical results.
9. Modify the **main** to decide whether the results should be printed on the screen or written to a file.
10. Use a **stringstream** object to concatenate the arguments stored in argv. Then unstack the arguments directly to the required variables.

Exercise 4: *std::ostream manipulation*

1. In class **DumperSeries** change the **dump** method definition to be:

   ```
   virtual void dump(std::ostream & os) = 0;
   ```

   Implement the needed changes in class **PrintSeries** to use this generic **std::ostream** instead of simply **std::cout**.

2. In class **PrintSeries** provide a default parameter as **std::cout**:

   ```
   void dump(std::ostream & os = std::cout);
   ```

3. Append a method in class **DumperSeries** to specify the required precision

   ```
   virtual void setPrecision(unsigned int precision);
   ```

4. Use that precision in the **dump** method of **PrintSeries**.
5. In the file **dumper_series.hh** append the operator declaration:

   ```
   inline std::ostream & operator <<(std::ostream & stream, DumperSeries & _this) {
     _this.dump(stream);
     return stream;
   }
   ```

6. Now you can use any object inheriting from **DumperSeries** with ostreams in the main: instantiate an output filestream object **ofstream** to output the results of the dump directly to a file.

Exercise 5: *Series complexity*

1. Evaluate the global complexity of your program.
2. Modify the **Series** class to have the members

   ```
   unsigned int current_index;
   double current_value;
   ```

3. Use these members in **ComputeArithmetic** and **ComputePi** in order to prevent re-computation of the entire series each time a **DumperSeries** would call it. Factorize the code as much as possible (by writing generic behavior in the **Series** class directly)

4. How is the complexity now ?

5. In the case you want to reduce the rounding errors over floating point operation by summing terms reversely, what is the best complexity you can achieve ?

Exercise 6: ***Integral***

An Integral can be written as the limit of a Riemann sum:

$$\int_a^b f(x)dx = \lim_{N\to\infty} \sum_{i=1}^{N} f(x_i)\Delta x$$

where,

$$\Delta x = \frac{b-a}{N} \text{ and } x_i = a + i\Delta x$$

1. Create a class named **RiemannIntegral** which inherits from **Series** and implements the computation of the integral $\int_a^b f(x)dx$. This class will have for attributes a, b and f.

2. Modify the main file to implement three possibilities for f, $f : x \to x^3, f : x \to cos(x)$ and $f : x \to sin(x)$. The main file should also be modified in order to let the user choose the values of a and b.

3. Using your code, compute the integrals:

$$\int_0^1 x^3 dx, \int_0^\pi cos(x)dx, \int_0^{\frac{\pi}{2}} sin(x)dx$$

4. Do you get the values expected ? For each function, at which value of N do you get the value expected ($\sim$ 2 digits after the decimal point)?