Week 3 — *Introduction to Standard Library (STL)*

The goal of the present exercises is to discover the main usage of the STL structures
This learning necessarily goes through practicing.
Use the starting point from GIT for this exercise

Exercise 1: *std::string*

- Open the main.cc file. In the main function, create a string containing your name

```
std::string myName = "Guillaume";
```

- Construct a string with the first parameter passed to the program

```
std::string argument = std::string(argv[1]);
```

- Test the content of the string using the '==' operator

```
if (argument == myName){
  std::cout << "You have a nice name" << std::endl;
}
```

Exercise 2: *Stream based Input/Output*

C++ uses stream-based input/output library to handle input/output to files or memory streams. In this exercise, we will look in to file I/O and string I/O functionalities of standard library. For help on I/O library, please refer to C++ Reference - Input/Ouput library

- In the main function, open a file named `toto.txt` and write co-ordinates of a point (1.5, 1.0, 2.0) in it.

- Open the created file `toto.txt` in your favourite editor and change the co-ordinates to (3.72, 2.13, 1.0). In the main function, read the `toto.txt` with modified co-ordinates and store the new co-ordinates in separate variables `x, y, z`. **Hint :** Use `std::stringstream`

Exercise 3: *std::array and std::vector*

- In the main function, create an array (`std::array`) which stores the co-ordinates of the point from the file `toto.txt`. Print the content of the array on the screen.

- In the main function, create a vector (`std::vector`) of double.

- In order to append items to the vector you can use the method 'push_back'. Append 100 values ranging from 0 to 2π (π is defined under the standard name M_PI)

- Print the size of the vector to screen

- If you want to resize the vector you can use the 'resize' method. If you enlarge the vector, what values will be contained in the extra part of the vector ? What is the 'assign' method doing?
  For more help, please refer to C++ Reference - Vector

- Loop over the values of vector to construct a vector named `sin_vect` which contains both pi value as well as its corresponding sinus value. **Hint:** Use `std::pair` and `std::make_pair` to create `sin_vect`
  C++ Reference - Pair

```
std::vector< std::pair<double, double> > sin_vect;
```

- Iterate over `sin_vect` to generate a file containing two columns (CSV format) with the values of pi and its corresponding sinus value. Scientific notation, with 20 digits of precision is expected. The 1st and 2nd element of a pair is accessed through functions `first` and `second`

```
std::cout << sin_vect[0].first;  // prints the first element at 0th posiiton of
    vector
std::cout << sin_vect[0].second; // prints the second element at 0th position of
    vector
```

Exercise 4: *Memory Management*

In C++, the programmer has to manage two kinds of memory: the **stack** and the **heap**. Variables allocated on the stack are short-lived: they die at the end of the block in which they were created. However the stack is very fast, which is good for performance. If we want data to live longer than a block, we allocate on the heap, which is much larger than the stack, but also much slower.

In this exercise, we will manipulate objects on the stack and the heap, and exploit dynamic allocation for persistent data.

- Fill in the blanks in the starting point code so that `values` is an array in the stack of size 10.
- Change the previous code so that values is on the heap. Why is there a compile error? Correct it.
- Is the memory allocated on the heap freed at the end of the program? What should you change?

We now look at data persistence:

- Implement the function `stack_allocation` to return the address of a `std::vector` of `n int`s on the stack.
- Implement the function `heap_allocation` to return the address of a `std::vector` of `n int`s on the heap.
- Why is the program failing?
- After removing the stack allocation to make the program run successfully, is the memory correctly managed?

Manipulating raw pointers as we have done until now is to be avoided in modern C++: we use the following alternatives when we can:

- `int values[10]` → `std::array<int, 10> values`
- `int * values = new int[10]` → `std::vector<int> values(10)`
- `std::vector<int> * heap_allocation(int n)`
  → `std::unique_ptr<std::vector<int> > heap_allocation(int n)`

Now you can redo the first two questions with `std::array` and `std::vector`, and replace the `new` in the `heap_allocation` function by `std::make_unique`. Is the memory correctly managed?

Finally, what do you need to change to the file `third.cc` to make it compile, so that the memory managed by the pointer `unique` is correctly transferred to `owner`?

Exercise 5: *std::map and std::set*

- In the main file, define a data type `Point` which is a alias for `std::array<double, 3>`.

```
using Point = std::array<double, 3>;
```

- In the main function, create a map from `std::string` to `Point`.

  ```
  std::map<std::string, Point> map;
  ```

- Let us use this map to store relative coordinates of planets. In order to append items to the map you can use the '[ ]' operator:

  ```
  map["sun"] = origin_coordinates;
  ```

- Add the earth, with the associated coordinates being (1.,0.,0.) in astronomical unit.

- What happens if you try to append an item with an already-exsiting entry ?

- Prevent this issue using *find* method of map:

  ```
  std::map<std::string, Point>::iterator it = map.find("earth");
  ```

- Loop over the entries of the map and output the entire content to screen

- Create another map from `std::string` to `Point`. This map will contain coordinates of 4 planets

  ```
  mercury : 0.25, 0., 0.
  earth   : 1.0, 0., 0.
  jupiter : 5.0, 0., 0.
  sun     : 0., 0., 0.
  ```

- Create two sets (`std::set`) which contains only keys of the above created two maps. C++ Reference - Set

- Use function `std::set_intersection` to get the keys which are common in both sets. Store these keys in an another set. C++ Reference - Set Intersection

- Now, based on the above set of intersected keys create a new map which contains the data corresponding to those keys only.

Exercise 6: *Standard Algorithms*

In this exercise, we will use algorithms of the Standard Template Library. They are generic implementation of useful, general-purpose algorithms (e.g. `std::sort`, `std::find`, etc.) that work with the STL containers, but also with user-defined types. Here, we will review some algorithms that are commonly used in scientific computing.

Algorithms are often combined with lambda-functions: they are anonymous functions, i.e. they do not have a name. They can easily be passed as arguments to algorithms to modify their behavior (e.g. a function which gives the order relationship of two elements for `std::sort`, a custom reduction operation for `std::reduce`).

For this exercise, look at the reference page https://en.cppreference.com/w/cpp/algorithm. Most algorithms take as parameters a `begin` and `end` iterators.

- Create a `std::vector<double>` of size `n`. Using the algorithm `std::fill` (not the member function `vector::fill` !), fill the vector with zeros. Check the expected result with a range-for loop.

- Look at the documentation for the `std::iota` algorithm. Apply it to the vector with a starting value of 1 (include `<numeric>`).

- Create a lambda function that takes a reference to `double` as an argument and squares it. Verify the expected result on a simple `double` variable. You can look up the syntax in the course notes.

- Using a range-for loop, apply this lambda to the previously initialized vector. Check the expected result.

- Replace the range-for loop with the `std::for_each` algorithm.

- We now want to compute the sum of all the elements in the vector. We can accomplish this by capturing a variable in a lambda function:

  - Create the variable that will hold the result

  - Create a lambda-function capturing that variable and summing its argument into it.

  - Using `std::for_each`, apply the lambda and check the sum result against $s = n(n+1)(2n+1)/6$.

  - If you get the wrong answer, check how you capture the result variable. Is it by-value or by-reference?

- Look up the documentation for `std::accumulate`. Do the previous question without a lambda function (be careful about the type of the `init` value).

- Finally, with the same vector containing squares of natural numbers, use the answer to the questions above to compute an approximate value of π using the sum of inverse squares, in reverse order (see the documentation of `std::vector`), without any loop.