# Advanced Programming Tutorial
# Worksheet 9: The C++ Standard Library

**Keywords:** containers, iterators, algorithms

## Prepare!

Before coming to the tutorial, and so that the time is enough to solve the exercise, revise the slides of the lecture unit "The Standard Library" and answer the following questions:

- ☐ How do you know if you are using headers from the C or from the C++ standard library?
- ☐ What STL algorithms do you know? Write an example of applying an algorithm of your choice to a container of your choice.
- ☐ What are ranged algorithms? What are views?
- ☐ Read about Conway's Game of Life. What are the rules?
- ☐ Look into the file `game-of-life.cpp`: What does each function do?

In this worksheet, we will look for and apply a few STL algorithms, trying to write short, clear, and reliable code. You may not already know all the algorithms we will use here from the lecture, but you already know how to use one if you find it. Therefore, we will also try to first find algorithms to simplify the tasks we have in mind.

## Exercise 1: Conway's Game of Life

Conway's Game of Life[1] is a simple automaton, which shares many features with a numerical simulation of a transient system. The game is played automatically on a 2D grid of cells, each cell being either "dead" (value 0) or "alive" (value 1). The grid (population) evolves in time, and in every time step we apply a few simple rules to each cell (quoted by Wikipedia):

1. Any live cell with two or three live neighbors survives.
2. Any dead cell with three live neighbors becomes a live cell.
3. All other live cells die in the next generation. Similarly, all other dead cells stay dead.

We provide a skeleton implementation in `game-of-life.cpp`, based on a matrix data structure defined in `Matrix.h`. The main algorithm is described in `main()`, and the above rules are applied in `evolve`. In our implementation, we are imposing a periodic boundary condition (similar to the games Pacman and Snake). To set these periodic boundary conditions, we are using a "ghost layer" around the real domain.

1. Look at the file `Matrix.h`. What data can it store and what methods does it provide? What does the already implemented constructor do?

[1] Conway's Game of Life in Wikipedia: `https://en.wikipedia.org/wiki/Conway%27s_Game_of_Life`

2. We want to implement an overloaded variant of our constructor, which does not generate random values, but it fills the matrix with a given `uniform_value`. However, we want to do this using an STL algorithm. Look at cppreference[2] for an algorithm that can fill a container with a given value and apply it.

3. Implement the method `set_row`, so that it copies a provided vector to a given row of the matrix. Find an algorithm that copies values.

4. Look at the file `game-of-life.cpp` and identify the main steps of the algorithm. Note an implementation-specific detail documented in the function `evolve`.

5. Apply an algorithm to accumulate (calculate the sum of) all the neighbors of a cell and store it to the `sums` matrix.

6. Apply the evolve function to each element in `sums`.

7. **Idea:** Apply different initial patterns and try to converge to a steady-state (see Wikipedia).

8. **Idea:** Replace the `.begin()/.end()` versions of these algorithms with equivalents from the C++20 ranges.

9. **Idea:** Replace the printing with a plot, using an external visualization library, such as sciplot[3] or matplotplusplus[4].

To speed-up the computation during development, you may want to remove the call to `sleep_for()` in `main()`. You may also want to experiment with different domain sizes and number of epochs.

## C++ Core Guidelines

**SL.1** Use libraries wherever possible

**SL.2** Prefer the standard library to other libraries

**SL.con.2** Prefer using STL vector by default unless you have a reason to use a different container

**SL.str.2** Use `std::string_view` or `gsl::span<char>` to refer to character sequences

**SL.io.50** Avoid `endl`

## Epilogue

We do use STL algorithms all the time, if they can provide any operation we otherwise would have to implement ourselves. The main limitation is typically what is offered by the C++ standard version we have to follow. Current research on the execution policies aims to automatically run the algorithms on different executors (multi-core CPU, GPUs of different architectures, etc).

*Chair of Scientific Computing in Computer Science – Technical University of Munich*
*Last updated on January 8, 2025*

[2]Algorithms library on cppreference: `https://en.cppreference.com/w/cpp/algorithm`
[3]sciplot on GitHub: `https://github.com/sciplot/sciplot`
[4]matplotplusplus on GitHub: `https://github.com/alandefreitas/matplotplusplus`