C++ Programming

Exercise Sheet 8 / Bonus
Secure Software Engineering Group
Philipp Schubert
philipp.schubert@upb.de

June 18, 2021

Solutions to this sheet are due on 02.07.2021 at 16:00. Please hand-in a digital version of your answers via PANDA at `https://panda.uni-paderborn.de/course/view.php?id=22691`.
Note: If you copy text or code elements from other sources, clearly mark those elements and state the source. Copying solutions from other students is prohibited.

On this exercise sheet, you will deal with file IO and implement a small simulation. You are free to implement your own useful data structure—a hash table—as a bonus exercise sheet. You can achieve 16 points + 16 bonus points for the optional exercise.

Exercise 1.
Programming a two dimensional cellular automaton — *Game of Life*. Cellular automatons are often used to simulate real-world scenarios. For instance, one could say that weather forecast is only a three dimensional cellular automaton with a bunch of simple rules. The *Game of Life* is a particular cellular automaton consisting of a two dimensional grid. The grid, in turn, comprises fields. Each field, to which we refer to as a cell, can be in one of two states: the state **1** (alive) or **0** (dead). The automaton is called *Game of Life* because the state of a cell might change from one generation to another according to some rules. The *Game of Life* allows arbitrary many generations that are computed iteratively.

The rules for computing the next generation, based on the current generation, are as follows (each cell interacts with its eight neighbors, which are the cells that are horizontally, vertically or diagonally adjacent):

1. A dead cell becomes alive if exactly 3 of its adjacent cells are alive.
2. A living cell dies if fewer than 2 or more than 3 of its adjacent cells are alive.
3. In any other case the state of a cell remains the same.

As all cells positioned at the boarders of the grid do not have 8 adjacent cells, you do not have to consider them for the sake of simplicity. Keep that in mind when you iterate the cells of the current generation in order to compute the next generation. You may use two nested **std::vector**s of boolean variables to store the state of a grid as shown in the code snippet below.

```
// this generates an 8 x 10 grid and initializes all cells to '0'
std::vector<std::vector<bool>> grid(8, std::vector<bool>(10, 0));
```

```
// printing the grid
for (const auto &row : grid) {
  for (const auto &cell : row) {
      std::cout << cell << ' ';
  }
  std::cout << '\n';
}
// individual cells can be accessed by using operator[]
std::cout << "grid at position [1][2] is: " << grid[1][2] << '\n';
```

You may wish to have a look at `https://en.wikipedia.org/wiki/Conway's_Game_of_Life` for a more detailed description.

a) Implement a function **std::vector<std::vector<bool>> read_grid(const std::string &filename);** that reads a a grid from a text file and parses it into a **std::vector** of **std::vector**s of **bool** values. (4 P.)

b) Next, implement another function
std::vector<std::vector<bool>>
game_of_life(const std::vector<std::vector<bool>> &grid, const size_t N); that returns a grid obtained by "waiting" (computing) **N** generations for the input grid. Hint: Use two temporary "grid" variables. Compute the cells for the next generation grid by checking the rules for the current generation and write new states to the next generation grid. When you have completed the computation for the next generation, use **std::vector**'s member function **swap()** to swap the contents of the two temporary variables and proceed until you have computed the N-th generation. (8 P.)

c) You know what next, implement a function
void write_grid(const std::string &filename, const std::vector<std::vector<bool>> &grid); that writes a grid to a text file. (3 P.)

d) Test all of your functions by reading the grid from `initial_grid.txt` that looks a bit like a snowman and compute the grid that is obtained by waiting (computing) the 10th generation. Write the result back to a file. (1 P.)

Exercise 2.

This is an optional exercise that is worth the equivalent of 16 points:
In this exercise, you will implement a simple hash table. Do not worry, we will split this task into little subtasks. A hash table H is a data structure that stores values v_i that are associated with keys k_i. We assume that the keys are unique. A value in H can be efficiently accessed by applying a hash function $h : \mathbb{S} \to \mathbb{N}$ to a key (with $\mathbb{S}$ the set of all possible strings which is our key domain in this exercise). $h(k)$ tells us where the value that is associated with the key $k \in \mathbb{S}$ is stored in memory. Since the application of a hash function to a key is a computation that only requires a constant amount of time, H is a data structure allowing to access arbitrary values in constant time, on average. Thus, a hash table is one of the most used data structures in practice.

a) First, you have to provide some code that should make up your hash table H. It is probably a good idea to make H a class, since H is a more sophisticated data type. Do so and create a class **HTable**. (1 P.)

b) In this exercise, we want to restrict ourselves to only associate **std::string**s with variables of an arbitrary type. For that reason, make **HTable** a class template that receives one template parameter **T**. (1 P.)

c) In order to store the elements in **HTable** in an easily accessible manner, provide a data member **std::vector<std::pair<std::string, T>> data** that allows us to store key-value pairs. Additionally, provide a data member **std::vector<bool> positions_in_use** to keep track of the used positions in your hash table. (1 P.)

d) Provide a constructor **HTable(size_t size);** that initializes the member variables **data** and **positions_in_use** to hold **size** elements. (1 P.)

e) Now, you need to provide a function member h that receives a key (a **std::string** value in our case) and returns a positive integer that shall be used as the index / position at which the key-value pair associated with that key must be stored. Use the hash function shown in code listing Listing 1 to do the job. The function turns a **std::string** into a natural number $n \in \{0, .., size - 1\}$ which is exactly what we need. (1 P.)

f) Next, implement a member function **bool insert(const std::string &key, const T &value);**. **insert()** has to compute the index at which the key-value pair should be stored by using the previously implemented function **hash()**. Insert the key-value pair at the position obtained by calling **hash()** on **insert**'s parameter **key** and set the corresponding bit in **positions_in_use** to mark this position as used. At last, let **insert()** return the boolean value **false**. (1 P.)

g) We have made a mistake. What if two (different) keys are accidentally mapped to the same index? This is called a hash collision and the probability of such a collision grows with an increasing number of entries stored in the hash table. A strategy to resolve this problem is required. We resolve this problem by making H a hash table with so-called "linear probing". That is: we check if the calculated position in **data** is not in use. If it is not in use, we insert the key-value pair at this very position. If the computed position is already in use, we linearly probe if one of the next positions in **data** is not in use and insert the data into the next "empty" position. Let the **insert()** function return **true** when you have to use linear probing while inserting a key-value pair. As a hash table gets filled with more and more data, you have to do more and more linear probing because of hash collisions. The access behavior of a hash table slowly changes from constant time to linear time in the worst case. If you have reached the end of **data** and still have not found an empty position, continue to check for empty positions starting from the beginning. If there is no empty position at all throw a **std::runtime_error** to notify users of your hash table that the table is full. (3 P.)

h) Now, let us provide a function to get data out of our hash table. Implement **T& get(const std::string &key);** to hash the **key** and retrieve the **value** associated with that key at position $h(key)$. Again, due to possible hash collision (recall how we have inserted data) it might be possible that the calculated entry does not contain the corresponding value we are looking for. Therefore, you have to compare **key** (the formal parameter of **get()** with the **key** stored at the calculated position. If both keys match (use **==** for the comparison) we have found the correct entry and can return the corresponding value. If the keys do not match, use linear probing and linearly check for the next entries and return the value as soon as you find both keys matching. If you reach the end of the underlying **std::vector** start at the beginning and throw a **std::runtime_error** if the key cannot be found in the hash table. (4 P.)

i) Overload **friend std::ostream operator<< (std::ostream &os, const HTable &h);** to print all key-value pairs. (1 P.)

j) Implement a function **void erase(const std::string &key);** that deletes the entry that corresponds to the key **key**. You can delegate this erase to **data**'s member function erase. Do not forget to set the bit in **positions_in_use** to zero to mark the place as free. (1 P.)

k) At last, implement a function **void clear();** that deletes all entries in your hash table. The number of elements that can be stored in **data** should remain the same. (1 P.)

```
size_t hash(const std::string &key) {
  size_t hash_val = 5381; // have a nice prime number
  for (const char c : key) {
    hash_val = hash_val * 33 + c;
  }
  // since 'hash_val' is an unsigned type, we can just ignore overflows
  // (overflow is well-defined for unsigned types)
  // because we need a value between 0 and size-1, we take the remainder of division by table's size
  return hash_val % data.size();
}
```

Listing 1: A hash function that turns out to be quite efficient.

Now relax, you have done a really good job so far.