Bonusaufgaben zum C/C++-Praktikum

Speicherverwaltung

TECHNISCHE UNIVERSITÄT DARMSTADT

Übungsblatt 2 – Sommersemester 2025

Hinweise zur Abgabe:
Verwende als Grundlage für die Bearbeitung die im Moodle bereitgestellte Vorlage. Beachte auch die Hinweise auf dem ersten Bonuszettel.

Allgemeines zum Format der Vorlage ab dieser Bonusaufgabe:
In der Vorlage findest du in der Regel die Gerüste der zu implementierenden Klassen. Diese Deklarationen enthalten schon die in den Aufgabenstellungen beschriebenen öffentlichen Schnittstellen, d. h. die `public` Methoden, gegen die deine Implementierung dann auch getestet werden wird. Jede dieser Methoden enthält auch eine minimale Definition, damit der Code kompiliert. Deine Aufgabe ist es, diese durch entsprechende sinnvolle Defintionen zu ersetzen. Dazu musst Du in der Regel auch private Attribute in der Klasse deklarieren und kannst auch gerne, wenn es dir sinnvoll erscheint, weitere (private) Methoden definieren.

In der Vorlage sind die minimalen Definitionen der Methoden alle direkt in der hpp-Datei implementiert. Du kannst die Definitionen alle oder zum Teil auch gerne in die entsprechende cpp-Datei verschieben. Denke dann nur daran, dass im Header dann nur noch die Deklaration stehen darf.

Conway's Gane Of Life
In dieser Aufgabe wirst du Conway's Game Of Life implementieren. Game Of Life ist ein zweidimensionaler zellulärer Automat. Es handelt sich dabei um ein Null-Spieler-Spiel. Eine Interaktion mit dem Spiel findet nur über die Festlegung des Startzustandes statt, danach läuft das Spiel automatisch ab. Game Of Life findet auf einem diskreten, zweidimensionalen Spielfeld (Zeilen und Spalten) statt. Der Spielverlauf ist iterativ, also rundenbasiert. Jeder Punkt auf diesem Spielfeld wird als Zelle oder Kolonie bezeichnet. Regeln bestimmen vor jeder Iteration, ob eine solche Kolonie in der nächsten Iteration „lebendig" oder „tot" sein wird.

Grafische Visualisierung
Die Vorlage sollte kompilierbar sein, auch wenn die Tests – natürlich – noch fehlschlagen. Sollte die Vorlage nicht kompilieren, kann es daran liegen, dass Probleme mit der zur Verfügung gestellten grafischen Ausgabe (siehe optionale Aufgabe 4) auftreten. Dort ist auch beschrieben, was bei verschiedenen Betriebssystemen getan werden muss, damit dies lauffähig wird. Alternativ kann auch einfach die Vorlage `02_memory_without_SFML` verwendet werden, die diese grafische Ausgabe nicht enthält.

Aufgabe 1: [S] Kolonie

In der Vorlage findest du in `Colony.hpp` das Gerüst der Klasse `Colony`. Diese stellt eine einzelne Zelle im Game Of Life dar.

1a) Grundgerüst (4 Punkte)

Implementiere den Konstruktor

```
Colony::Colony(bool initial_state);
```

Dieser nimmt als Argument den Anfangszustand der Kolonie, wobei `true` für „lebendig“ und `false` für „tot“ steht.

Implementiere zusätzlich die Methode

```
bool Colony::getState() const;
```

die den aktuellen Zustand der Kolonie zurück gibt und die Methode

```
void Colony::evolve();
```

die den aktuellen Zustand auf den nächsten Zustand setzt, der zuvor über den Aufruf von `Colony::calculateNextState` (siehe nächster Aufgabenteil) bestimmt wurde.[1]

1b) Nächster Zustand (2 Punkte)

Jede Kolonie oder Zelle im Game Of Life hat acht direkte Nachbarn. Es gelten folgende Regeln:

1. Eine lebendige Kolonie wird in der nächsten Iteration nur dann lebendig sein, wenn sie von zwei oder drei anderen lebendigen Kolonien umgeben ist. Sonst wird sie wegen Unter- bzw. Überbevölkerung sterben.
2. Eine tote Kolonie wird in der nächsten Kolonie aufgrund von Expansion in der nächsten Iteration lebendig sein, wenn sie genau drei lebendige Nachbarn hat.
3. Liegt eine Kolonie am Rand des Spielfeldes, soll dieser als tot angesehen werden. Eine Kolonie am Rand kann also maximal fünf lebendige Nachbarn haben und eine Kolonie in einer Ecke des Spielfeldes kann maximal drei lebendige Nachbarn haben.

Implementiere die Methode

```
void Colony::calculateNextState(unsigned int alive_neighbors);
```

die mit der Anzahl der aktuell lebenden Nachbarn aufgerufen wird und daraus den Folgezustand bestimmt und intern speichert, so dass dieser mit dem nächsten Aufruf von `Colony::evolve` als neuer Zustand übernommen werden kann.

[1] Die Trennung in die Bestimmung des nächsten Zustands sowie das tatsächliche Setzen des neuen Zustandes ist für die hier folgende Variante der Implementierung dieses Spiels notwendig.

Aufgabe 2: [S] Game Of Life

In dieser Aufgabe wirst du die Klasse `GameOfLife` implementieren, die alle Kolonien des Spielfeldes enthält und sich um das Iterieren der Zustände kümmert. Die Deklaration befindet sich in der Datei `GameOfLife.hpp`.

Aus Übungszwecken ist hier das Format vorgegeben, in der die Kolonien, die das Spielfeld darstellen, gespeichert werden. Daher enthält diese Klasse neben den öffentlichen Methoden auch schon das private Attribut `colonies_` vom Typ

```
typedef std::vector<std::vector<Colony*>> Universe;
```

das das „Universum“ bzw. Spielfeld repräsentiert.[2] Dabei soll der äußere Vektor die y-Richtung (Zeilen) und der innere Vektor die x-Richtung (Spalten) darstellen. D. h. der Zugriff auf ein Element $(x,\ y)$ des Spielfeldes soll mit

```
colonies.at(y).at(x)
```

bzw.

```
colonies[y][x]
```

erfolgen.

Auch hier kannst Du bei Bedarf der Klasse gerne weitere Attribute hinzufügen.

2a) Konstruktor und Destruktor (5 Punkte)

Implementiere

```
GameOfLife::GameOfLife(size_t x_size, size_t y_size, unsigned int prob);
```

so dass ein Spielfeld mit `x_size` Spalten und `y_size` Zeilen erstellt wird. Jedes Element des Spielfeldes soll eine neue `Colony` sein, die auf dem Heap gespeichert wird. Dabei soll die Wahrscheinlichkeit dafür, dass eine Kolonie am Anfang lebendig ist, gleich `prob` (in Prozent) sein. `prob` kann dabei Werte von 0 bis 100 annehmen.

Schaue dir hierzu die Funktionen `std::rand` und `std::srand` an, mit denen man zufällige Zahlen generieren kann.[3]

Stelle im Destruktor

```
GameOfLife::~GameOfLife();
```

sicher, dass alle auf dem Heap allozierten Ressourcen auch wieder freigegeben werden.

Implementiere zusätzlich einen zweiten Konstruktor

```
GameOfLife::GameOfLife(Universe& colonies);
```

[2] Diese Darstellung des Spielfeldes ist in dieser Aufgabe bewusst so gewählt, damit in der Lösung mit der *vector*-Klasse der Standard Template Library sowie der manuellen Speicherverwaltung gearbeitet wird. Tatsächlich ist diese Darstellung des Spielfeldes etwas verschwenderisch und ineffizient: Dadurch, dass das Spielfeld als Vektor von Vektoren abgebildet wird, sind für beispielsweise 100×100-Spielfeld 101 Allokationen erforderlich, anstelle von einer Allokation, die notwendig wäre, wenn man das Spielfeld in einem einzigen zusammenhängenden Vektor speichern würde. Und zur Erzeugung der *Colony*-Einträge sind hier 10 000 Allokationen notwendig, obwohl ein *Colony*-Objekt auch direkt in den Vektoren erzeugt werden könnte (und dabei auch noch kürzer als ein Zeiger ist). Neben der aufwändigen Erzeugung des Spielfeldes ist im weiteren Verlauf auch der Datenzugriff ineffizient, da drei Zeiger-Dereferenzierungen notwendig sind, um auf eine bestimmte *Colony* zuzugreifen.

[3] Die Initialisierung des Zufallszahlengenerators sollte einmal beim Start des Programmes geschehen, z. B. in der `main`-Funktion. Die Initialisierung darf nicht vor jedem Aufruf von `std::rand` durchgeführt werden, da dann die Zufälligkeit tatsächlich zum Teil verloren geht und die Tests nicht mehr funktionieren. Die Tests kümmern sich selbst um eine entsprechende Initialisierung.

der die Member-Variable `colonies_` mit dem übergebenen Wert initialisiert. Das (als Referenz) übergebene Argument soll nach dem Funktionsaufruf „leer“ sein, d. h. `colonies.size() == 0`. (Überlege dir, weshalb!)[4]

Die „Rule of three“ erfordert es, bei dieser Klasse auch einen eigenen Kopierkonstruktor und den Kopie-Zuweisungsoperator zu implementieren, und nicht die automatischen Implementierungen des Kompilers zu nutzen.[5] (Überlege dir, warum dies der Fall ist!) Für diese Übung musst du nur den Kopierkonstruktor

```
GameOfLife::GameOfLife(const GameOfLife& other);
```

implementieren. Für den Kopie-Zuweisungsoperator können wir einfach mit

```
GameOfLife& operator=(const GameOfLife&) = delete;
```

dafür sorgen, dass dieser nicht automatisch erzeugt wird. Dies schränkt zwar die allgemeine Verwendbarkeit der Klasse ein, aber im Rahmen dieser Übung müssen keine Zuweisungen mit Objekten der Klasse `GameOfLife` erfolgen.

2b) Getter (1 Punkt)

Implementiere die Getter-Methoden

```
size_t GameOfLife::getXsize() const;
size_t GameOfLife::getYsize() const;
const Universe& GameOfLife::getColonies() const;
```

die die Breite und Höhe des Spielfeldes sowie eine const-Referenz auf die Kolonien des Spielfeldes zurückgeben.

2c) Iteration (4 Punkte)

Implementiere nun die Methode

```
void GameOfLife::iterate();
```

In dieser Methode soll der Zustand für die nächste Iteration berechnet werden. Dazu muss für jede Kolonie die Anzahl der lebendigen Nachbarn ermittelt werden und mithilfe dieses Wertes der nächste Zustand berechnet und am Ende auch gesetzt werden.

Hinweis: Ist eine Kolonie am Rand des Spielfeldes, soll der Rand als „tot“ angesehen werden.

[4] Hier könnte man den Konstruktor auch so definieren, dass dieser eine RValue-Referenz erwartet, `GameOfLife(Universe&& colonies)`, und das Argument in die Member-Variable „moven“.

[5] Den vom Kompiler erzeugten move-Konstruktor und die entsprechende Überladung des Zuweisungsoperators können wir hingegen in diesem Fall weiterhin nutzen.

Aufgabe 3: [S] Lesen und Schreiben in Dateien

In dieser Aufgabe wirst du zwei Methoden implementieren, mit denen ein momentaner Zustand eines `GameOfLife` in eine Textdatei geschrieben und wieder daraus gelesen werden kann. Dazu ist ein einheitliches Format nötig. Dies ist in der in der Vorlage enthaltenen Datei `gol.txt` zu sehen. In der ersten Zeile stehen, mit einem Leerzeichen getrennt, `x_size` und `y_size`. Darunter stehen in Zeilen-Spalten-Form die Zustandswerte der einzelnen Kolonien. Dabei steigt der Wert der x-Koordinate von links nach rechts an. Der Wert der y-Koordinate steigt hingegen von oben nach unten an. Eine 1 heißt, dass die Kolonie lebendig ist, eine 0 bedeuted, dass die Kolonie tot ist. Also würde der Input

```
3 3
010
011
100
```

dem Zustand in Abbildung 1 entsprechen.

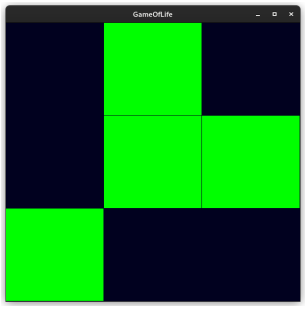

Abbildung 1: Game Of Life Zustand

3a) Schreiben (2 Punkte)

Implementiere die Methode

```
void GameOfLife::writeToFile(std::string filename);
```

die den aktuellen Zustand des `GameOfLife` wie oben beschrieben in die Datei mit dem Pfad `filename` schreibt.

Hinweis: Schaue dir hierzu die Methoden der Klasse `std::fstream` an. Mithilfe des entsprechenden Konstruktors lässt sich eine txt-Datei öffnen und anschließend dem erzeugten `fstream`-Objekt mit dem `<<`-Operator neuen Text hinzufügen. Wenn die Variable des `fstream`-Objekts ihren Gültigkeitsbereich verlässt, werden alle möglicherweise noch gebufferten Daten vollständig geschrieben und die Datei geschlossen.

3b) Lesen (3 Punkte)

Implementiere nun die statische Member-Funktion

```
static GameOfLife GameOfLife::readFromFile(std::string filename);
```

Diese soll eine Datei mit dem oben erklärten Format einlesen und daraus ein `GameOfLife`-Objekt erstellen und dieses zurückgeben. Wenn das Dateiformat nicht stimmt, soll die Methode einen `std::runtime_error` mit einer entsprechenden Beschreibung werfen.

Hinweis: Schaue dir hierzu am besten die Funktion `std::getline` an.

Aufgabe 4: [S] Ein Game Of Life ausführen (Optional)

Grafische Darstellung

Die meisten Spiele haben eine sogenannte „Main Loop“. Diese ist eine Schleife, die den Ablauf des Spiels solange regelt, bis eine Endbedingung erreicht ist. In der „Main Loop“ wird also der Spielablauf gesteuert. Bei Game of Life ist dieser relativ simpel. Zuerst muss für jede Kolonie der nächste Zustand ermittelt werden, und dann muss der Zustand von jeder Kolonie auf den Folgezustand gesetzt werden.

Wir stellen für diese Aufgabe eine einfache Visualisierung mit SFML (Simple and Fast Multimedia Library) zur Verfügung, die auch die „Main Loop“ implementiert. Diese ist in der Klasse `Visualizer` implementiert, die eine Instanz von `GameOfLife` als Attribut enthält. Wir verwenden hier das SFML CMake Template. Um SFML zu verwenden, musst du die entsprechenden Abhängigkeiten installieren. Im Folgenden findest du eine kurze Installationsanleitung. Außerdem sind weitere Informationen hier zu finden: `https://www.sfml-dev.org/learn.php`.

Wenn die Installation nicht funktionieren sollte, steht im Moodle-Kurs eine alternative Vorlage ohne SFML zur Verfügung, so dass du deine Implementation der restlichen Aufgaben trotzdem testen kannst.

Windows SFML läuft nativ auf Windows mittels der WIN32-API. Hier sind also keine zusätzlichen Konfigurationen nötig.

Linux Unter Linux (Debian-based OS) müssen Abhängigkeiten mit dem folgenden Befehl installiert werden:

```
sudo apt update
sudo apt install \
    libxrandr-dev \
    libxcursor-dev \
    libudev-dev \
    libopenal-dev \
    libflac-dev \
    libvorbis-dev \
    libgl1-mesa-dev \
    libegl1-mesa-dev \
    libdrm-dev \
    libgbm-dev
```

Bei anderen Linux Distributionen mit anderen Paket-Managern ist ein ensprechend analoger Befehl notwendig.

Mac OS Die Installation von SFML unter Mac OS ist deutlich komplizierter als unter Windows oder Linux. Zunächst müssen die SFML-Bibliotheken mittels des Befehls

```
brew install sfml
```

installiert werden. Der Installationspfad ist dann mittels `brew info sfml` ermittelbar. Dieser wird von nun an mit `<pfad>` bezeichnet. Nun müssen folgende Zeilen in `CMakeLists.txt` eingefügt werden:

```
include_directories(<pfad>/include)
find_library(SFML_SYSTEM sfml-system PATHS <pfad>/lib)
find_library(SFML_WINDOW sfml-window PATHS <pfad>/lib)
find_library(SFML_GRAPHICS sfml-graphics PATHS <pfad>/lib)
find_library(SFML_AUDIO sfml-audio PATHS <pfad>/lib)
find_library(SFML_NETWORK sfml-network PATHS <pfad>/lib)
target_link_libraries(ProjektName sfml-system sfml-window sfml-graphics sfml
    -audio sfml-network)
```

Zuletzt nach dem Einfügen dieser Zeilen in CLion mit rechts auf `CMakeLists.txt` klicken und dann **Reload CMake Project** auswählen.

Um die Visualisierung zu verwenden, muss ein Objekt der Klasse `Visualizer` mit einem Objekt der Klasse `GameOfLife` und einem Integer `unsigned int scale` initialisiert werden. Der Wert von `scale` bestimmt dabei die Seitenlänge der Quadrate, die in der Visualisierung Kolonien darstellen. Mit der Methode `void Visualizer::run()` kann dann das Spiel ausgeführt werden.

4a)

In der Vorlage ist eine Datei Namens `gol.txt`, die einen Startzustand für ein `GameOfLife` enthält, das letztendlich in einem stabilen, eingeschwungenen Zustand endet. Erstelle in der `main`-Funktion einen `Visualizer` mit eben diesem Startzustand und rufe dessen „Main Loop“ auf.