Goethe-Center for Scientific Computing (G-CSC)
Goethe-Universität Frankfurt am Main

Einführung in die Programmierung (EPR)

(Übung, Wintersemester 2014/2015)

Dr. S. Reiter, M. Rupp, Dr. A. Vogel, Dr. K. Xylouris, Prof. Dr. G. Wittum

Aufgabenblatt 8 (Abgabe: Fr., 23.01., 9:29h online)

Conways "Spiel des Lebens"

Das Spiel des Lebens nach Conway dreht sich um ein rechteckiges Spielbrett auf dem $n \times m$ Zellen in einem regelmäßigen Gitter angeordnet sind. Eine jede solche Zelle hat entweder den Status *"lebendig"* oder *"tot"*. Anhand spezieller Regeln kann sich der Status einer jeden Zelle abhängig vom Status seiner Nachbarzellen dabei im Verlauf des Spiels von *lebendig* zu *tot* oder umgekehrt von *tot* zu *lebendig* ändern. Als Nachbarn einer Zelle `X` werden dabei die 8 Zellen (außer `X`) betrachtet, die in einem 3×3 Block mit Zentrum `X` liegen. Die Nachbarschaft einer Zelle `X` ist im Folgenden schematisch dargestellt. Nachbarn von `X` werden dabei mit `N` bezeichnet:

```
. . . . .
. N N N .
. N X N .
. N N N .
. . . . .
```

Man betrachte im Folgenden ein Spielbrett mit *lebendigen* Zellen (markiert durch den Buchstaben `'O'`) sowie *toten* Zellen (markiert durch das Zeichen `'.'`), z.B.

```
......
..O...
...O..
.OOO..
......
......
```

Die Regeln für die Änderung des Zellstatus lauten dann wie folgt:

1. Hat eine *tote* Zelle genau 3 lebendige Nachbarn, so wird die Zelle belebt. In der folgenden Runde ist der Status der Zelle also *lebendig*.

2. Hat eine *lebendige* Zelle weniger als 2 oder mehr als 3 lebendige Nachbarn, so stirbt die Zelle. In der nächsten Runde hat sie also den Status *tot*.

3. Trifft weder Regel 1 noch Regel 2 zu, so bleibt der aktuelle Zellstatus in der nächsten Runde erhalten.

Dabei ist folgende wichtige Vorgehensweise zu beachten: Der Statusübergang der Zellen von einer Runde in die nächste findet bei allen Zellen gleichzeitig statt. Um den neuen Status einer Zelle zu berechnen, muss also die Nachbarkonfiguration der unmodifizierten aktuellen Runde betrachtet werden.
Im Folgenden sind einige Runden des *Spiel des Lebens* ausgehend von obiger Startkonfiguration (`turn 0`) beispielhaft dargestellt.

```
......   ......   ......   ......   ......
..O...   ......   ......   ......   ......
...O..   .O.O..   ...O..   ..O...   ...O..
.OOO..   ..OO..   .O.O..   ...OO.   ....O.
......   ..O...   ..OO..   ..OO..   ..OOO.
......   ......   ......   ......   ......
turn 0   turn 1   turn 2   turn 3   turn 4
```

Aufgabe 1 (4+4+4+4+4+4+15+1 Punkte)
Betrachten Sie das folgende Klasseninterface:

```
class GameOfLife{
    public:
        GameOfLife(int rows, int cols);
        ~GameOfLife();
        void clear();
        void set(int row, int col, int value);
        void set(int row, int col, const char* values);
        int get(int row, int col);
        void print();
        void advance();
};
```

Das Spielfeld soll als ein Feld von Integers im *privaten* Abschnitt der Klasse verwaltet werden. Der Wert 0 entspricht dabei einer *toten* Zelle, alle anderen Werte entsprechen einer *lebendigen* Zelle.

a) Implementieren Sie den Konstruktor `GameOfLife(int rows, int cols)`. Dieser soll den für ein Spielfeld mit `rows` Reihen und `cols` Spalten benötigten Speicher automatisch allozieren. Legen Sie die für den Zugriff auf das Spielfeld nötigen Variablen im `private` Bereich der Klasse an.
Implementieren Sie außerdem den Destruktor `~GameOfLife()`, der den für das Spielfeld allozierten Speicher wieder frei gibt.

b) Implementieren Sie die Methode `void clear()`, die alle Einträge des Spielfelds auf `0` setzt. Rufen Sie `clear()` in Ihrem Konstruktor auf, um das neu angelegte Spielfeld zu initialisieren.

c) Implementieren Sie die Methode `void set(int row, int col, int value)`. Diese soll den Wert der über `row` und `col` indizierten Zelle des Spielfelds auf `value` setzen.

Liegen `row` oder `col` außerhalb des gültigen Bereichs, soll die Methode terminieren ohne einen Wert zu setzen.

d) Implementieren Sie `void set(int row, int col, const char* values)`, die mehrere Einträge in Zeile `row` gleichzeitig setzt.
Dabei werde für `i=0,...,strlen(values)-1` die Methode `set(row, col+i, v[i])` aufgerufen, mit:

$$\texttt{v[i]} := \begin{cases} 0, & \text{falls } \texttt{values[i]=='.'} \\ 1, & \text{sonst.} \end{cases}$$

Hinweis: Der Befehl `strlen` wird in der Bibliothek `cstring` definiert.

e) Implementieren Sie die Methode `int get(int row, int col)`, die den Wert der über `row` und `col` indizierten Zelle des Spielfelds zurückliefert. Liegen `row` oder `col` außerhalb des gültigen Bereichs, so soll die Methode `0` zurückgeben.

f) Implementieren Sie die Methode `void print()`, die den Inhalt des Spielfelds Zeile für Zeile im Terminal ausgibt. Dabei soll für jede *tote* Zelle das Zeichen `'.'` und für jede *lebendige* Zelle der Buchstabe `'O'` ausgegeben werden. Für ein 2×2 Spielfeld, in dem ausschließlich die Zelle in der linken oberen Ecke *lebendig* ist, sollte die Ausgabe entsprechend wie folgt aussehen:

```
O.
..
```

g) Implementieren Sie die Methode `void advance()`, die über die in der Einleitung beschriebenen Regeln den Übergang des Spielfelds von der aktuellen in die nächste Runde implementiert. Achten Sie darauf, dass der Status von Zellen in der nächsten Runde ausschließlich vom eigenen Status sowie dem Status der Nachbarzellen aus der alten Runde abhängen darf.

h) Testen Sie Ihre Implementierung mittels folgendem Code:

```
#include <iostream>
using namespace std;

//... your code here ...

void PlayGameOfLife(GameOfLife& g, int iterations){
    g.print();
    for(int i = 0; i < iterations; ++i){
        g.advance();
        cout << endl;
        g.print();
    }
}

int main(){
```

```
    GameOfLife g(10, 10);
    g.set(2, 2, ".O..O.");
    g.set(3, 2, "O.OO.O");
    g.set(4, 2, ".O..O.");
    g.set(5, 2, ".O..O.");
    g.set(6, 2, "O.OO.O");
    g.set(7, 2, ".O..O.");
    PlayGameOfLife(g, 5);
    return 0;
}
```

Hinweis: Nach 5 Iterationen sollte sich wieder das Startmuster ergeben.

Aufgabe 2 (4+4+4+1 Punkte)
Für diese Aufgabe soll das Spielfeld des *Spiel des Lebens* als torusförmig (oder periodisch) betrachtet werden. Verlässt man ein torusförmiges Spielfeld auf der rechten Seite, so betritt man es gleichzeitig auf der linken Seite und umgekehrt. Verlässt man ein solches Spielfeld auf der unteren Seite, so betritt man es gleichzeitig an der oberen Seite und umgekehrt. Gegenüberliegende Ränder werden also miteinander identifiziert.
Sei `g` eine Instanz der `GameOfLife` Klasse mit einem torusförmigen Spielfeld mit `rows` Zeilen und `cols` Spalten. Dann gilt entsprechend obiger Erklärung für ein Indexpaar $(a, b) \in \mathbb{Z}$:

$$\forall i, j \in \mathbb{Z} : \texttt{g.get(a,b) = g.get(a+i*rows, b+j*cols)}$$

a) Erweitern Sie die Klasse `GameOfLife` aus Aufgabe 1 um eine private Membervariable `bool m_periodic`. Diese soll im Konstruktor der `GameOfLife` Klasse auf `false` gesetzt werden. Implementieren Sie außerdem eine Methode `void set_periodic(bool periodic)`, über die die Membervariable `m_periodic` gesetzt werden kann, sowie eine Methode `bool is_periodic()`, über die der Wert der Variable `m_periodic` abgefragt werden kann.

b) Passen Sie die Methode `int get(int row, int col)` der `GameOfLife` Klasse so an, dass im Fall von `is_periodic()==true` die in Aufgabe 2 einleitend beschriebenen Regeln für torusförmige Spielfelder gelten. Gibt man beim Aufruf einen Zeilen- oder Spaltenindex außerhalb des Spielfelds an, so soll dieser so umgerechnet werden, dass die auf einem torusförmigen Spielfeld entsprechende Zelle mit Zeilenindex $0 \leq r <$ `rows` und Spaltenindex $0 \leq c <$ `cols` gefunden und deren Wert zurückgegeben wird.
Im Fall von `is_periodic()==false` soll sich die Methode `int get(int row, int col)` weiterhin so verhalten wie in Aufgabe 1 gefordert.

c) Passen Sie die Methode `int set(int row, int col, int value)` der `GameOfLife` Klasse so an, dass im Fall von `is_periodic()==true` die in Aufgabe 2 einleitend beschriebenen Regeln für torusförmige Spielfelder gelten. Gibt man beim Aufruf einen Zeilen- oder Spaltenindex außerhalb des Spielfelds an, so soll dieser so umgerechnet werden, dass die auf einem torusförmigen Spielfeld entsprechende Zelle mit Zeilenindex $0 \leq r <$ `rows` und Spaltenindex $0 \leq c <$ `cols` gefunden und deren Wert beschrieben wird.

Im Fall von `is_periodic()==false` soll sich die Methode `int set(int row, int col, int value)` weiterhin so verhalten wie in Aufgabe 1 gefordert.

d) Testen Sie Ihren Code mit folgender `main()`-Funktion (`PlayGameOfLife` wie in A1):

```
int main(){
    GameOfLife g(5, 5);
    g.set_periodic(true);
    g.set(-3, -3, ".O.");
    g.set(-2, -3, "..O");
    g.set(-1, -3, "OOO");
    PlayGameOfLife(g, 20);
    return 0;
}
```

Hinweis: Nach 20 Iterationen sollte sich wieder das Startmuster ergeben.

Aufgabe 3 (2+4+1)
Abschließend soll der in der Vorlesung eingeführte Pseudo-Zufallszahlengenerator genutzt werden, um den Status der einzelnen Zellen zufällig zuzuweisen.

a) Implementieren Sie den Zufallszahlengenerator aus der Vorlesung (Kap. 5, Folie 41).

b) Erweitern Sie die `GameOfLife` Klasse um eine Funktion `void randomize(Zufall z)`, die den übergebenen Zufallszahlengenerator nutzt, um den Einträgen des Spielfelds einen zufälligen Status zuzuweisen. Dabei soll für jede Zelle eine Zahl gezogen werden. Ist diese Zahl durch 2 teilbar, so soll die Zelle den Status *tot* erhalten, ist die Zahl hingegen nicht durch 2 teilbar, so soll die Zelle den Status *lebendig* erhalten.
Hinweis: Der Modulo-Operator `%` ist in diesem Zusammenhang sehr nützlich.

c) Testen Sie Ihren Code mit folgender `main()`-Funktion (`PlayGameOfLife` wie in A1):

```
int main(){
    GameOfLife g(12, 60);
    g.randomize(Zufall(3));
    PlayGameOfLife(g, 150);
    return 0;
}
```

Information zur Abgabe: Bitte schicken Sie Ihre Lösung als `*.cpp` / `*.h` Dateien an Ihren Übungsgruppenleiter per Mail. Wichtig ist, dass Sie Ihre Lösung freitags vor 9:30h abgeschickt haben. Bitte schicken Sie nur den Quellcode, nicht jedoch ihr gesamtes Eclipse-Projekt oder ausführbare Dateien.