Dr. U. Köthe, P. Schill *Heidelberg, 18. Januar 2017*

Übungsblatt 11

Aufgabe 11.1 Polynomklasse [12 Punkte]

Die Datei `polynomial.cpp` auf Moodle enthält den Rumpf einer Klasse `Polynomial` und eine Reihe von Tests. Ein Polynom ist wie üblich definiert als

$$a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n$$

Der maximale Exponent n heißt *Grad* des Polynoms (engl. *degree*). Die Koeffizienten a_i sind in `Polynomial`-Objekten in einer Array-Variable `a_` gespeichert. Ist `p` ein Objekt vom Typ `Polynomial`, so kann man mit `p[i]` (eckige Klammern) auf den i-ten Koeffizienten zugreifen. Mit `p(x)` (runde Klammern) wird das Polynom für das Argument x ausgerechnet (eine effiziente Implementation dafür ist das Horner-Schema, siehe Übung 2.2(d)). Falls Ihnen die Polynomaddition und -multiplikation nicht klar ist, informieren Sie sich im Internet.

Vervollständigen Sie die Klasse so, dass alle Tests durchlaufen, und geben Sie die modifizierte Datei ab. Implementieren Sie alle fehlenden Funktionen als Member-Funktionen der Klasse. *Hinweis:* Eine (einfachere) Aufgabe dieser Art kann Teil der Klausur sein.

Aufgabe 11.2 Templatisierung der Punktklasse [12 Punkte]

In der Vorlesung haben wir behandelt, wie man eine Klasse in ein Template umwandelt. Sie sollen dies hier anhand der bereits bekannten Klasse `Point` üben, einschließlich der arithmetischen Operationen aus Übung 10.1. Verwenden Sie die Datei `point_template.cpp`, die Sie auf Moodle finden, als Ausgangspunkt und geben Sie die modifizierte Datei ab. Gehen Sie in folgenden Schritten vor:

(a) Vervollständigen Sie die Funktion `test_Point_double()` in der Datei `point_template.cpp`, um aussagekräftige Tests für die existierende Klasse `Point` zu implementieren. Diese Funktion beginnt mit einer Typdefinition

```
typedef Point P;
```

und verwendet den Punkttyp danach immer über den Namen `P`. Auf diese Weise können Sie die Testfunktion später auf ein `Point`-Template umstellen, indem Sie nur diese Typdefinition (also nur eine einzige Zeile) ändern. Implementieren Sie zwei Tests für jede Funktion. Sie können dabei die Tests der arithmetischen Operationen aus Übung 10.1 wiederverwenden. Denken Sie daran, auch krumme Zahlen zu testen!

Führen Sie die Funktion `test_Point_double()` ab jetzt nach jeder Änderung aus. Dann bemerken Sie sofort, wenn Sie bei der Umstellung einen Fehler gemacht haben, und können den letzten Schritt rückgängig machen.

(b) Fügen Sie einen lokalen Typ

```
typedef double CoordinateType;
```

in die Klasse `Point` ein. Ersetzen Sie den Typ `double` überall dort, wo er in der Klasse `Point` als Typ der Koordinaten benutzt wird, durch `CoordinateType`. Dadurch ändert sich nichts an der Funktionalität – wir haben nur einen neuen Namen für den Typ `double` verwendet. Die Tests müssen also weiterhin funktionieren!

(c) Fügen Sie die Template-Deklaration vor der Klasse ein:

```
template <typename CoordinateType>
class Point
{ ... };
```

Da der `CoordinateType` jetzt durch die Template-Deklaration definiert ist, müssen Sie das `typedef` aus Schritt (b) nun wieder entfernen. Ändern Sie außerdem die Typdefinition am Anfang von `test_Point_double()` so, dass das `Point`-Template jetzt mit `double` als Koordinatentyp verwendet wird:

```
typedef Point<double> P;
```

Kommentieren Sie alle freien Funktionen und die korrespondierenden Tests vorübergehend aus. Die übrigen Tests müssen nun wieder funktionieren!

(d) Templatisieren Sie die freien Funktionen: Entfernen Sie nach und nach die Kommentare, die Sie in Schritt (c) vorübergehend eingefügt haben. Achten Sie dabei darauf, immer nur an einer Funktion (und den zugehörigen Tests) zu arbeiten, bis `test_Point_double()` wieder durchläuft. Erst dann wird die nächste Funktion (samt deren Tests) wieder einkommentiert.

Wandeln Sie jede freie Funktionen in ein Template um, zum Beispiel:

```
template <typename CoordinateType>
std::string to_string(Point<CoordinateType> const & p)
{ ... }
```

und passen Sie, wo notwendig, die Implementation an. Kontrollieren Sie durch regelmäßiges Ausführen der Tests, dass der Code in Ordnung ist.

Denken Sie bei der skaleren Multiplikation und Division (z.B. `operator*(Point p, double s)`) daran, dass als Typ des Faktors `s` bzw. Divisors `d` ebenfalls der `PixelType` eingesetzt werden muss.

(e) Implementieren Sie eine Funktion `test_Point_int()`, die mit der Typedefinition `typedef Point<int> P;` beginnt und das korrekte Verhalten des `Point`-Templates mit `int` als Koordinatentyp testet.

Aufgabe 11.3 Zelluläre Automaten - Conway's Game of Life [16 Punkte]

In dieser Aufgabe sollen Sie die `Image`-Klasse (siehe `image.hpp` auf Moodle) benutzen, um Conway's "Spiel des Lebens" zu implementieren. Dieses Spiel ist das bekannteste Beispiel für einen *zellulären Automaten* und ein beliebter Sport unter Informatikern.

Zelluläre Automaten erweitern die Zustandsautomaten aus Übungsblatt 1, indem viele identische Automaten miteinander interagieren. Ein zellulärer Automat wird also spezifiziert durch

- eine Menge von Zellen (beim "Spiel des Lebens" bilden diese ein regelmäßiges Gitter der Größe $M \times N$, welches man sehr schön als Bild visualisieren kann) und
- einen Zustandsautomaten für jede Zelle (d.h. die Menge der Zustände, die jede Zelle annehmen kann, sowie die Übergangsfunktion vom Zeitschritt t nach $t + 1$).

Die Ereignisse, die zu Zustandsübergängen führen, kommen dabei nicht von außen, sondern ergeben sich aus der aktuellen Konfiguration der *Nachbarschaft* jeder Zelle zum Zeitpunkt t. Beim "Spiel des Lebens" kann jede Zelle genau zwei Zustände haben: "tot" (durch die Farbe weiß symbolisiert) und "lebend" (durch die Farbe schwarz symbolisiert). Die Nachbarschaft einer Zelle bilden die acht Zellen, die sich direkt um die Zelle herum befinden. Für die *Übergangsfunktion* gelten folgende Regeln:

- Eine lebende Zelle, die zum Zeitpunkt t zwei oder drei lebende Nachbarn hat, ist zum Zeitpunkt $t + 1$ weiterhin lebendig.
- Eine lebende Zelle mit weniger als zwei lebenden Nachbarn stirbt (Vereinsamung).
- Eine lebende Zelle mit mehr als drei lebenden Nachbarn stirbt (Überbevölkerung).
- Eine tote Zelle mit exakt drei lebenden Nachbarn wird wiederbelebt. Alle anderen toten Zellen bleiben tot.

Das Verblüffende an diesen einfachen Regeln ist, dass viele Anfangskonfigurationen zu sehr interessanten Evolutionsmustern führen. Die bekanntesten Muster haben sogar Namen: Blinker, Gleiter, Gleiterkanone und -fresser usw. Weitere Informationen finden Sie unter

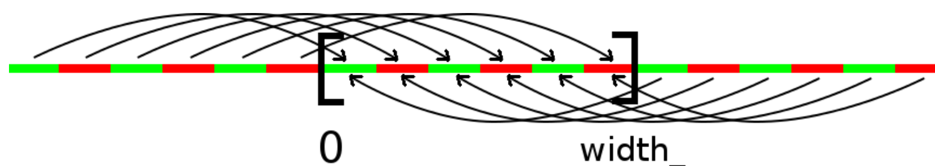
`http://de.wikipedia.org/wiki/Conways_Spiel_des_Lebens`
`http://www.mathematische-basteleien.de/gameoflife.htm`

Mit dem interaktiven Applet unter

`http://www.bitstorm.org/gameoflife/`

können Sie experimentieren und Konfigurationen ausprobieren.

(a) Da am Rand des Gitters nicht alle acht Nachbarn existieren, muss man eine *Randbehandlung* definieren. Am besten bewährt hat sich dabei die *periodische* Randbehandlung: Läuft man nach rechts aus dem Bild heraus, kommt man links wieder hinein und umgekehrt. Nach dieser Regel werden x-Koordinaten außerhalb des Bildes folgendermaßen in den gültigen Bereich transformiert:



Analog verfährt man mit den y-Koordinaten. Geben Sie (als Kommentar in `image.hpp`) mathematische Formeln für die Randbehandlung an. Implementieren Sie in der Klasse `Image` eine neue Memberfunktion

```
PixelType get_periodic(int x, int y) const
```

die den Zugriff mit periodischer Randbehandlung realisiert. Sie erhalten nun automatisch die periodische Randbehandlung, indem Sie `image.get_periodic(x, y)` anstelle von `image(x, y)` aufrufen.

(b) Erstellen Sie eine Datei `conway.cpp` und implementieren Sie eine freie Funktion

```
int count_alive_neighbors(Image const & image,
                          int x, int y)
```

die die Anzahl der lebenden Nachbarn der Zelle `(x, y)` unter Verwendung der periodischen Randbehandlung zurückgibt. Lebende Zellen sollen dabei durch den Grauwert 0, tote durch den Grauwert 255 dargestellt sein.

(c) Implementieren Sie die Funktion

    Image conway_step(Image const & image)

die die Übergangsfunktion auf alle Zellen des gegebenen Bildes anwendet und ein neues Bild mit der resultierenden Konfiguration zurückgibt.

(d) Da jede Zelle nur ein Pixel groß ist, kann man die aktuelle Konfiguration auf hochauflösenden Bildschirmen nicht mehr erkennen. Schreiben Sie eine Funktion

    Image scale_image(Image const & image, int scale)

die das Eingabebild `image` um den Faktor `scale` vergrößert. Dazu soll jedes Pixel in x- und y-Richtung `scale`-mal wiederholt werden.

(e) Auf Moodle finden Sie die Datei `conway_init.pgm`. Lesen Sie die Datei mit `readPGM()` ein und wenden Sie 100 mal `conway_step()` an. Skalieren Sie das Ergebnis um den Faktor 5 und testen Sie mittels `assert()`, dass Ihr Ergebnis mit der Datei `conway_test.pgm` (ebenfalls auf Moodle) übereinstimmt.

(f) Starten Sie wieder von `conway_init.pgm` und speichern Sie das 5-fach skalierte Spielfeld nach jeder Iteration mittels `writePGM()` unter den Namen `conway_sim_000.pgm`, `conway_sim_001.pgm`, … `conway_sim_010.pgm`, `conway_sim_011.pgm` … Achten Sie auf die führenden Nullen bei der Nummerierung (vergleiche dazu Aufgabe 6.1) – andernfalls werden die Bilder später eventuell in falscher Reihenfolge eingelesen. Erstellen Sie aus den Einzelbildern ein Video `simulation.mpg`. Dazu können Sie z.B. das Programm `convert` aus der ImageMagick-Installation verwenden:

```
convert -delay 6 conway_sim_*.pgm simulation.mpg
```

Der Parameter `-delay 6` legt die Abspielgeschwindigkeit fest. ImageMagick können Sie unter Linux einfach mit `apt-get` installieren, Installationsprograme für alle Betriebssysteme finden Sie unter `http://www.imagemagick.org/script/binary-releases.php`.

(g) Erstellen Sie ein eigenes Initialbild `my_conway.pgm`, das eine Startkonfiguration mit möglichst interessanter Evolution enthält. Sie können dieses Bild in C++ erzeugen und mit `writePGM()` exportieren, oder ein beliebiges Zeichenprogramm verwenden. Falls das Zeichenprogramm das PGM-Format nicht unterstützt, können Sie Ihr Bild mit ImageMagick `convert` nach PGM wandeln. Erzeugen Sie nun wie in (f) ein Video `my_simulation.mpg` mit der zeitlichen Entwicklung Ihres Bildes.

Geben Sie die modifizierte Datei `image.hpp` sowie die Dateien `conway.cpp`, `simulation.mpg`, `my_conway.pgm` und `my_simulation.mpg` ab.

Bitte laden Sie Ihre Lösung spätestens bis 25. Januar 2017, 9:00 Uhr in Moodle hoch.