Dr. U. Köthe *Heidelberg, 11. Januar 2017*

Übungsblatt 10

Aufgabe 10.1 Arithmetik für die Punktklasse [12 Punkte]

In der Vorlesung haben wir behandelt, wie man arithmetische Operationen für eigene Datentypen implementiert. Sie sollen dies in dieser Aufgabe anhand der bereits bekannten Klasse `Point` üben. Benutzen Sie als Ausgangspunkt die Datei `point.cpp`, die Sie auf Moodle finden, und geben Sie die modifizierte Datei ab. Die Tests sollen in einer Funktion `test_Point()` implementiert werden, die dann in `main()` aufgerufen wird.

(a) Implementieren Sie eine Funktion, um zwei Punkte im Sinne der Vektoraddition (also elementweise) zu addieren

```
Point operator+(Point p1, Point p2)
{ ... }
```

und schreiben Sie drei Tests für diese Funktion. *Elementweise* bedeutet, dass der x-Wert des Ergebnisses die Summe der x-Werte der beiden Eingabepunkte ist, ebenso der der y-Wert, zum Beispiel:

```
Point p(2.0, 3.0), q(4.0, 5.0);
Point r = p+q;
assert(r == Point(6.0, 8.0));
```

Benutzen Sie hier und in den weiteren Teilaufgaben auch "krumme" Werte für die Koordinaten – ansonsten könnten die Tests Bugs übersehen, die z.B. durch versehentliches Runden während der Berechnung entstehen.

Implementieren und testen Sie die Vektorsubtraktion mit `operator-()` ebenso.

(b) Implementieren Sie entsprechende Funktionen für die elementweise Multiplikation und Division von zwei Punkten. Schreiben Sie je drei Tests, zum Beispiel:

```
Point p(2.0, 3.0), q(4.0, 5.0);
Point r = p*q;
assert(r == Point(8.0, 15.0));
```

(c) Implementieren Sie Funktionen, um einen Punkt zu skalieren:

```
Point operator*(Point p, double s)
{ ... }

Point operator*(double s, Point p)
{ ... }
```

Im Unterschied zu (b) haben diese Funktionen zwei verschiedene Argumenttypen (`Point` und `double`), wobei der Skalierungsfaktor bei der ersten Version rechts und bei der zweiten Version links vom $*$-Zeichen angegeben wird. Implementieren Sie je drei Tests, zum Beispiel:

```
Point p(2.0, 3.0);
Point r = 3.0*p;
assert(r == Point(6.0, 9.0));
```

(d) Implementieren und testen Sie die Division mit einem skalaren Wert

```
Point operator/(Point p, double s)
{ ... }
```

(wir beschränken uns hier auf den Fall, dass das Skalar rechts vom `/`-Zeichen steht).

(e) Implementieren und Testen Sie die unäre Negation

```
Point operator-(Point p)
{ ... }
```

die den Punkt am Ursprung spiegelt, zum Beispiel

```
Point p(2.0, 3.0);
Point r = -p;
assert(r == Point(-2.0, -3.0));
```

Aufgabe 10.2 Bildklasse [16 Punkte]

In der Vorlesung haben wir die Klasse `Image` behandelt, mit der man 2-dimensionale Bilder speichern und bearbeiten kann. Auf Moodle finden Sie die Datei `image.hpp`, die einen Rumpf der Klassenimplementation enthält.

(a) Vervollständigen Sie den Code in `image.hpp` überall dort, wo `"IHR CODE HIER"` steht. Die Funktionalität, die Sie jeweils implementieren sollen, wird in Kommentaren beschrieben. Das PGM-Format wird in der Vorlesung und in der Wikipedia unter `https://de.wikipedia.org/wiki/Portable_Anymap` erklärt (wir verwenden speziell die Variante "Portable Graymap ASCII" mit der Kennung "P2"). Legen Sie außerdem für die folgenden Teilaufgaben eine Datei `image.cpp` an, die `image.hpp` inkludiert.

(b) Erzeugen Sie ein Bild mit der Größe `width=4` und `height=3`. Testen Sie mittels `assert()`, dass das Bild die gewünschte Größe hat. Testen Sie dann in einer zweifach geschachtelten Schleife, dass alle Pixel den Wert '0' haben. Testen Sie außerdem, dass die Funktion `to_string(image)` den erwarteten String zurückgibt.

(c) Füllen Sie das Bild mit einem Schachbrettmuster, d.h. abwechselnd mit den Grauwerten '0' (schwarz) und '255' (weiß). Das erste Pixel in der linken oberen Ecke soll schwarz sein. Testen Sie das Ergebnis mittels `assert()` und `to_string(image)`.

(d) Exportieren Sie das Bild aus (c) mit der Funktion `writePGM()` in das File `board4x3.pgm`. Öffnen Sie dieses File in einem Editor und überprüfen Sie, dass der Inhalt den Erwartungen entspricht. Legen Sie in Ihrem Programm ein zweites Bild an, in das Sie das File `board4x3.pgm` mittels `readPGM()` wieder einlesen. Testen Sie mit `assert()`, dass dieses Bild mit dem Originalbild aus (c) übereinstimmt.

(e) Schreiben Sie eine Funktion

```
Image chessboard(unsigned int width, unsigned int height,
                 unsigned int square_size)
```

die ein Schachbrett-Bild der angegebenen Größe erzeugt. Der Parameter `square_size` gibt dabei an, wie groß die einzelnen Schachfelder sein sollen. Das heißt, ein Quadrat der Größe `square_size*square_size` aus schwarzen Pixeln (Wert '0') soll sich oben links im Bild befinden, daneben und darunter entsprechende Quadrate aus weißen Pixeln (Wert '255') usw. Testen Sie mittels `assert()`, dass der Aufruf `chessboard(4, 3, 1)` das Schachbrett aus (c) reproduziert.

(f) Erzeugen Sie ein Schachbrett mit dem Aufruf `chessboard(400, 300, 20)` und geben Sie es in das File `board400x300.pgm` aus. Prüfen Sie mit einem Bildbetrachter, der das PGM-Format unterstützt (z.B. 'IrfanView' unter Windows, 'display' aus der imagemagick-Installation unter Linux, 'gimp' unter allen Betriebssystemen), dass der Inhalt des Files korrekt ist. Lesen Sie das File dann mit `readPGM()` wieder ein und testen Sie die Übereinstimmung mit dem Originalbild.

(g) Schreiben Sie eine Funktion
`Image invert_image(Image const & image)`
die das gegebene Bild invertiert, d.h. jeden Pixelwert `p` durch den Pixelwert `255-p` ersetzt. Wenden Sie diese Funktion auf das Schachbrett aus (f) an und exportieren Sie das Ergebnis als `board400x300-inverse.pgm`. Vergewissern Sie sich mit einem Bildbetrachter, dass die schwarzen und weißen Felder jetzt genau vertauscht sind.

(h) Auf Moodle finden Sie das Bild `lena.pgm`. Lesen Sie dieses Bild ein und invertieren Sie es. Exportieren Sie das invertierte Bild als `lena-inverse.pgm`.

Geben Sie die vervollständigte Datei `image.hpp` sowie die Datei `image.cpp` und die erzeugten Bilddateien ab.

Hinweis: Wir verwenden hier den Pixeltyp `uint16_t`, obwohl `uint8_t` für die Aufgabe eigentlich ausreichen würde. Unsere Wahl sichert, dass alle Compiler die Pixelwerte als Zahlen behandeln. Der Typ `uint8_t` hingegen wird von manchen Compilern als Zeichentyp interpretiert, was zu Fehlern in `to_string()` und `readPGM()` führen würde. Natürlich könnte man das Problem durch relativ einfache Implementierungstricks in diesen Funktionen lösen, wir wollen diese zusätzlichen Schwierigkeiten aber in der Aufgabe umgehen.

Aufgabe 10.3 Kaleidoskop [12 Punkte]

In dieser Aufgabe sollen Sie den Umgang mit Klassen im Allgemeinen und der `Image`-Klasse im Besonderen üben, indem Sie Funktionen zum Spiegeln eines Bildes implementieren. Geben Sie die Lösung in einer Datei `kaleidoscope.cpp` ab. Diese Datei soll die vervollständigte Datei `image.hpp` aus der vorigen Aufgabe inkludieren, um die bereits vorhandene Funktionalität der Bildklasse wiederzuverwenden.

(a) Implementieren Sie die Funktion

```
Image mirror_x(Image const & image)
```

die das gegebene Bild am rechten Rand spiegelt und gemeinsam mit dem Originalbild in ein doppelt so breites Bild einfügt, das dann zurückgegeben wird:

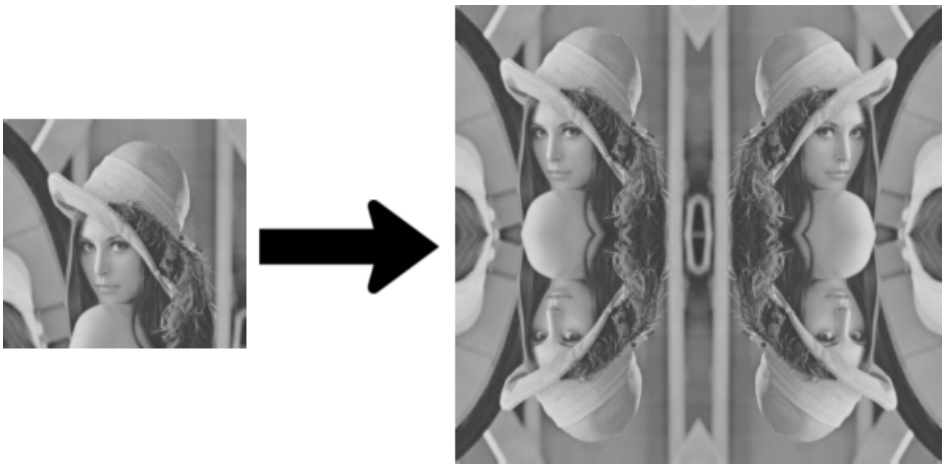
Implementieren Sie analog dazu die Funktion `mirror_y()`.

(b) Auf Moodle finden Sie die Bilddatei `lena.pgm`. Lesen Sie diese Datei mittels `readPGM()` in ein Bild vom Typ `Image` ein und wenden Sie `mirror_x()` bzw. `mirror_y()` auf dieses Bild an. Testen Sie mit `assert()`, dass Ihre Ergebnisse mit den Referenzbildern `mirror_x_test.pgm` bzw. `mirror_y_test.pgm` auf Moodle übereinstimmen.

(c) Implementieren Sie die Funktion

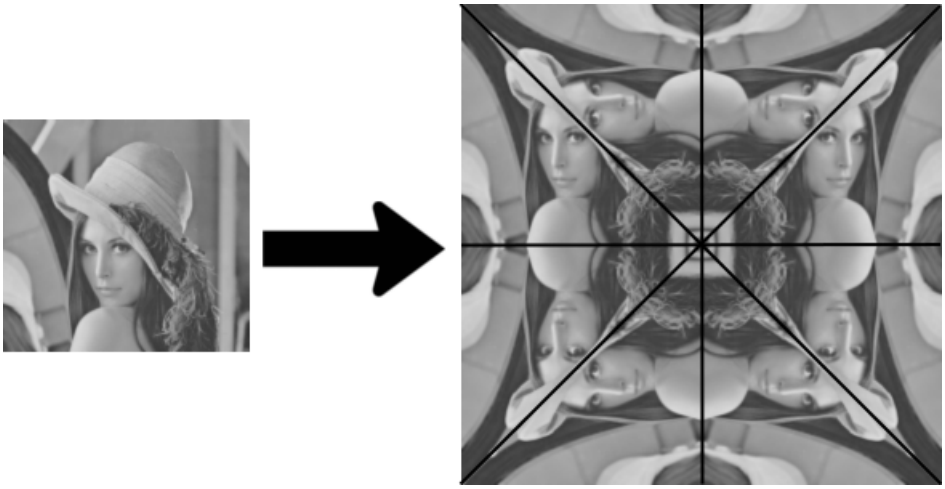
```
Image kaleidoscope4(Image const & image)
```

die das Bild mit Hilfe von `mirror_x()` und `mirror_y()` erst am rechten, dann am unteren Rand spiegelt und das Ergebnis zurückgibt:



(d) Fügen Sie in der Datei `kaleidoscope.cpp` einen Test für die Funktion `kaleidoscope4()` hinzu, die Ihr Ergebnis für `lena.pgm` mit dem Referenzbild `kaleidoskop4_test.pgm` auf Moodle vergleicht.

(e) Implementieren Sie analog zu (c) die Funktion `kaleidoscope8()`, die ein Kaleidoskop mit 8-facher Spiegelung wie im folgenden Bild erzeugt (die schwarzen Linien sind nur als Hilfslinien gedacht und sollen nicht gezeichnet werden):



Spiegeln Sie dazu das Eingabebild zunächst an der Diagonalen und benutzen Sie dann `kaleidoscope4()`. Das Resultat ist natürlich besonders wirkungsvoll, wenn das Eingabebild quadratisch war. Ihre Funktion soll aber so implementiert sein, dass Sie auch auf beliebige rechteckige Bilder angewendet werden kann.

(f) Fügen Sie in der Datei `kaleidoscope.cpp` einen Test für die Funktion `kaleidoscope8()` hinzu, die Ihr Ergebnis für `lena.pgm` mit dem Referenzbild `kaleidoskop8_test.pgm` auf Moodle vergleicht.

(g) Benutzen Sie ein Bildbetrachtungsprogramm, das das PGM-Format unterstützt (z.B. 'IrfanView' unter Windows, 'display' aus der imagemagick-Installation unter Linux, 'gimp' unter allen Betriebssystemen), um ein Bild Ihrer Wahl unter dem Namen `my_image.pgm` in dieses Format zu konvertieren. Wenden Sie `kaleidoscope8()` auf dieses Bild an und exportieren Sie das Ergebnis als `my_kaleidoscope.pgm`. Geben Sie die beiden Bilder ab. Wählen Sie ein Bild, das eine interessante Kaleidoskopdarstellung ergibt.

Bitte laden Sie Ihre Lösung spätestens bis 18. Januar 2017, 9:00 Uhr in Moodle hoch.