IMS
Institut für Mikroelektronische Systeme
Leibniz Universität Hannover

Programmierpraktikum Technische Informatik (C++)
Aufgabe 10

Hinweise
Abgabe: Stand des Git-Repositories am 14.7.2020 um 9 Uhr.
Die Dateien zur Bearbeitung dieser Aufgabe erhalten Sie, indem Sie die neue Aufgabe aus dem Aufgabenrepository in Ihr lokales mergen. Dies geschieht mit `git pull cpp2020 master` innerhalb Ihres Repositories. Die Lösungen committen Sie bitte in Ihr lokales Repository (`git commit -a` oder `git add` gefolgt von `git commit`) und pushen sie in Ihr Repository auf dem git-Server des Instituts (`git push`).

Im Folgenden soll im Verzeichnis `imageCompression` ein Programm vervollständigt werden, das Bilder im Targa-Format (.tga) komprimieren sowie dekomprimieren kann. Das Dateiformat sieht optional eine verlustfreie Kompression mittels einer Lauflängenkodierung vor.

Eine TGA-Datei beginnt immer mit einem Dateikopf, der in festgelegten Feldern Informationen über den Aufbau der übrigen Datei enthält. In den gegebenen Beispielen folgen auf den Dateikopf unmittelbar die Bilddaten als Abfolge der Farbwerte der Pixel. Zur Vereinfachung muss das Programm nur für Bilder mit Bilddaten vom Typ ARGB der Größe 32 Bit funktionieren. Alle zu diesem Zweck sinnvollen Abstraktionen sind bereits in der Vorgabe enthalten.

Die Klasse `TGAImage` repräsentiert ein **unkomprimiertes** TGA-Bild und speichert sämtliche erforderlichen Daten in den Membervariablen `header` und `data`. Die Klasse `TGAImageHeader` kapselt den relevanten, nicht-optionalen TGA-Dateikopf. Die **unkomprimierten** Bilddaten befinden sich in einem Vektor von Elementen des Typs `ARGB32`. Für alle Klassen aus der Vorgabe sind die **Streamoperatoren** so implementiert, dass diese eine binäre Ein- und Ausgabe ermöglichen. Für den Typ `ARGB32` sind darüber hinaus notwendige **Vergleichsoperatoren** überladen.

Das Ziel der Lauflängenkodierung besteht darin, aufeinander folgende Pixel mit identischem Farbwert zusammenzufassen und mit einem Zähler zu versehen, um Speicherplatz einzusparen. Daher enthalten die Bilddaten entsprechend kodierter Bilder zusätzlich zu den Farbwerten der Pixel (`ARGB32`) auch noch in unregelmäßigen Abständen Steuerbytes der Lauflängenkodierung (`char`). Die Bilddaten eines komprimierten TGA-Bildes beginnen immer mit einem Steuerbyte. Der Wert der unteren sieben Bits eines Steuerbytes bildet um eins inkrementiert einen Zähler, der dementsprechend minimal $0 + 1 = 1$ und maximal $(2^7 - 1) + 1 = 128$ betragen kann. Wenn das achte Bit („Most significant bit“ mit dem Wert $2^7 = 128$) eines Steuerbytes gesetzt ist, steht es für eine Wiederholung des darauffolgenden

Appelstraße 4 | 30167 Hannover
fon 0511 762-19691 | fax 0511 762-19694
www.ims.uni-hannover.de | info@ims.uni-hannover.de

Pixelwerts in der Länge des Zählerwerts. Auf diesen Pixelwert folgt dann direkt das nächste Steuerbyte. Wenn das Bit nicht gesetzt ist, gibt der Zähler Auskunft über die Anzahl der auf das Steuerbyte folgenden, unterschiedlichen Pixelwerte, bis das nächste Steuerbyte für die nächste Sequenz auftritt. Falls Unklarheiten bestehen, befindet sich im Abschnitt Lauflängenkodierung des Artikels `https://de.wikipedia.org/wiki/Targa_Image_File` eine anders formulierte Beschreibung.

Die Ausführung des Programms erzeugt insgesamt 12 Bilder im Verzeichnis `results`. Wenn Ihre Lösung korrekt funktioniert, sollten die resultierenden Bilder genau so aussehen wie die Eingabebilder im Ordner `examples`. Darüber hinaus sollte das Skript `check-results.sh`, das aus `diff`-Aufrufen besteht, keinerlei Ausgabe auf der Konsole bewirken. Die Korrektheit der **komprimierten** Dateien kann jedoch nicht zweifelsfrei durch einen Vergleich mittels `diff` überprüft werden, da das Verfahren zwar nur eine optimale, aber theoretisch mehr als eine gültige Kodierung zulässt.

Teilaufgabe 1 (2.5 Punkte)

Implementieren Sie in `imageCompression/src/runlengthencoding.cpp` die Funktion `encode`! Diese erhält als Parameter ein **unkomprimiertes** Bild vom Typ `TGAImage`, das mittels der beschriebenen Lauflängenkodierung komprimiert werden soll. Dabei soll das Ergebnis unmittelbar über den vorhandenen `std::ostream` in eine Datei geschrieben werden.

a) Iterieren Sie mit einem Iterator über die Pixel (`TGAImage::getData`) des Eingabebilds. Verwenden Sie `std::adjacent_find` zur Suche nach dem Beginn der nächsten Wiederholung von Farbwerten sowie `std::find_if`, um das Ende der Wiederholung zu finden. **Die Rechereche nach der Funktionsweise der Standard-Algorithmen auf cppreference.com ist Bestandteil der Aufgabe.**

b) Legen Sie im lokalen Gültigkeitsbereich der `encode`-Funktion eine Lambda-Funktion an, die über einen Variablennamen aufrufbar ist. Die Lambda-Funktion soll zwei Iteratoren (`auto`) und einen Wahrheitswert (`bool`) übergeben bekommen. Der Wahrheitswert sagt aus, ob die Funktion zur Behandlung einer Wiederholung aufgerufen wurde. Erzeugen Sie dementsprechend für den unkodierten Bereich zwischen den beiden Iteratoren eine gültige Kodierung. Sie können Steuerbytes mittels der `put`-Methode des `std::ostream` ausgeben. Die Ausgabe von Farbwerten (`ARGB32`) soll durch einen Aufruf von `std::copy` in Kombination mit einem `std::ostream_iterator` erfolgen.

   Hinweis:

   - Durch einen Konfigurationsfehler der bereitgestellten VirtualBox verwendet qtcreator einen veralteten C++-Standard für die Codeanalyse. Daher werden generische Lambdas mit Parametern vom Typ `auto` nicht als gültige Sprachkonstrukte erkannt. Sie können die falschen Fehlermeldungen im Editor

des qtcreator vermeiden, wenn Sie folgende Einstellung vornehmen: Unter „Tools“->„Options“->„Kits“->„Manual->Desktop (default)“ müssen Sie bei „Qt version“ die Option „None“ auswählen und „Ok“ drücken. Dann muss qtcreator beendet und neu gestartet werden. Ob die Änderung funktioniert hat, sehen Sie bei einem geöffneten Projekt unter „Tools“->„C++“->„C++ Code Model Inspector“->„Project Parts“->„General“. Dort muss bei „Language Version“ der Wert „CXX17“ stehen.

c) Rufen Sie die Lambda-Funktion aus Aufgabenteil `b` innerhalb ihrer Schleife aus Aufgabenteil `a` zielführend mit den in `a` ermittelten Iteratoren als Parameter auf, bis das Eingabebild vollständig verarbeitet ist.

Checkliste:
`std::adjacent_find`, `std::find_if`, `std::copy`, `std::ostream_iterator`

Teilaufgabe 2 (2.5 Punkte)

Implementieren Sie in `imageCompression/src/runlengthencoding.cpp` die Funktion `decode`! Sie soll ein lauflängenkodiertes Bild dekomprimieren und das Ergebnis als `TGAImage` zurückgeben. Das **komprimierte** Bild kann mit Hilfe des übergebenen `std::istream` gelesen werden.

a) Verarbeiten Sie den Stream des Eingabebilds, bis dieser keine Zeichen mehr enthält. Das erste nach dem Dateikopf gelesene Byte (`std::istream::get`) ist das erste Steuerbyte (`char`) der Lauflängenkodierung. Ermitteln Sie anhand des jeweils aktuellen Steuerbytes, ob dieses eine Wiederholung beschreibt sowie den dazugehörigen Zähler.

Hinweis:

- Die Methode `get` des `std::istream` hat einen Rückgabewert vom Typ `int`. Lassen Sie sich davon nicht verunsichern. Die Methode extrahiert trotzdem nur ein Byte aus dem Stream. Um Fehler zu vermeiden, können Sie gelesene Steuerbytes ebenfalls in einer `int`-Variable (oder `unsigned char` bzw. `uint8_t`) zwischenspeichern, damit die Werte im Bereich von 0 bis 255 liegen. Im Fall vom Typ `char` wäre es der nicht unbedingt günstige Bereich von -128 bis 127.

b) Schreiben Sie im lokalen Gültigkeitsbereich eine Lambda-Funktion namens `getPixel`, die einen Farbwert (`ARGB32`) aus dem Inputstream liest (`>>-Operator` ist überladen) und zurückgibt.

c) Vewenden Sie innerhalb Ihrer Schleife aus Aufgabenteil `a` in Abhängigkeit des jeweils vorliegenden Steuerbytes entweder `std::fill_n` oder `std::generate_n` in Kombination mit einem `std::back_inserter` und der Lambda-Funktion aus

Aufgabenteil `b`, um eine Sequenz zu dekodieren und in den Vektor des `TGAImage` einzufügen.

Checkliste:

`std::fill_n`, `std::generate_n`, `std::back_inserter`