Fachhochschule Nordwestschweiz
Hochschule für Technik

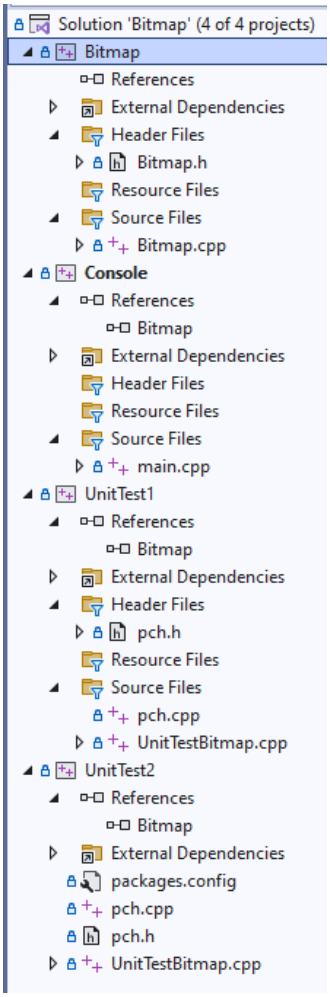
Programmieren in C++

Prof. Dr. Christoph Stamm christoph.stamm@fhnw.ch

Übung 1: Bitmap

Lernziele

- Sie repetieren den Stoff aus der Vorlesung: Struktur eines C++-Programms, einfache und strukturierte Datentypen, Zeiger, Referenzen, Arrays, Klassen, Konstruktoren/Destruktoren.
- Sie implementieren eine eigene Bitmap-Klasse, welche zur Speicherung von unkomprimierten Bildern im RGB- und ARGB-Format verwendet werden kann. Die Bilder haben die Dateiendung *.bmp und können von vielen Betriebssystemen und Bildprogrammen gelesen und dargestellt werden.
- Sie testen Ihre Implementierung mit zur Verfügung gestellten Unit-Tests.



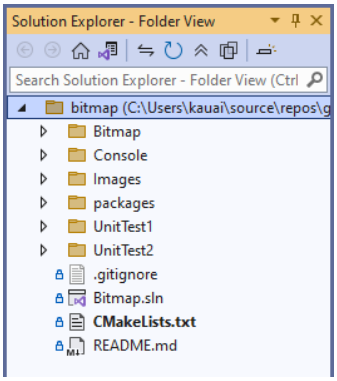
1 Projektstruktur

Für diese Übung gehen Sie am besten von nebenstehender Projektstruktur aus:

Die Solution enthält vier Projekte:

1. *Bitmap* ist eine statische Bibliothek. Sie enthält die Implementierung der Klasse `Bitmap`.
2. *Console* ist eine einfache Konsolenanwendung, welche Sie für kleine Tests und den Einsatz der Klasse `Bitmap` verwenden können. Diese Konsolenanwendung verwendet die statische Bibliothek *Bitmap*.
3. *UnitTest1* ist ein Projekt mit nativen Unit-Tests. Es enthält ein Set von Unit-Tests in *UnitTestBitmap.cpp* für die Klasse `Bitmap`. In Visual Studio handelt es sich bei dem UnitTest-Projekt um eine DLL. Diese DLL kann nicht direkt ausgeführt werden, sondern wird innerhalb des *Test Explorers* ausgeführt. Dazu nachfolgend mehr.
4. *UnitTest2* ist ein Projekt mit Unit-Tests, welche das Test-Framework von Google verwendet. Es enthält das gleiche Set von Unit-Tests wiederum in der Date *UnitTestBitmap.cpp* für die Klasse `Bitmap`. Diese Unit-Test sind vor allem für Benutzer von Linux und MacOS gedacht, können aber auch von Windows-Benutzern verwendet werden.

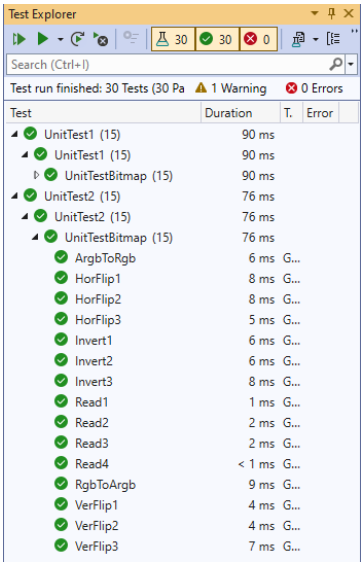
1.1 CMake und Visual Studio Solution



Die Projektstruktur ist so aufgebaut, dass Sie entweder die Datei «Uebung1.sln» in Visual Studio mit all darin enthaltenen Projekten und Projekteinstellungen öffnen können oder dass Sie den Solution-Ordner als *Folder* in Visual Studio öffnen können. Im letzteren Fall werden darin enthaltene CMake-Dateien «CMakeLists.txt» automatisch gelesen und von CMake ausgeführt. Es werden dann zwei *Targets* (*Console.exe* und *UnitTest2.exe*) erkannt, welche wie üblich gebildet und ausgeführt werden können.

Die Ordnerstruktur mit den CMake-Dateien kann auf allen Plattformen, welche CMake unterstützen, verwendet werden. Die meisten C++-IDEs unterstützen CMake und daher sollten Sie keine grösseren Schwierigkeiten haben, diese Übung in Ihrer bevorzugten IDE zu bilden und auszuführen.

1.2 Unit Tests



In diesem Abschnitt lernen Sie, wie Sie in Visual Studio native C++ Unit-Test-Projekt ausführen können. Beachten Sie, dass die vorgegebene *Solution* schon zwei fixfertige Unit-Test-Projekte enthält: *UnitTest1* verwendet das Unit-Test-Framework von Visual Studio und *UnitTest2* verwendet das Unit-Test-Framework von Google. Für Benutzer von CMake wird nur UnitTest2 verfügbar sein.

Sobald Sie die Solution neu kompiliert haben, können Sie die Tests ausführen lassen. Dazu gehen Sie im Menü auf *Test/Run All Tests* oder zum Debuggen der Tests wählen Sie *Test/Debug All Tests*. Selbstverständlich können Sie auch nur die im *Test Explorer* selektierten Tests ausführen. Das Fenster *Test Explorer* sollte beim Ausführen der Tests automatisch geöffnet werden. Falls nicht, so können Sie es über *Test/Test Explorer* öffnen. Die vorangegangene Abbildung zeigt einen Ausschnitt aus dem *Test Explorer*.

2 Windows Bitmap

Ein digitales Farbbild ist eine zweidimensionale Matrix von Farbpunkten. Die Matrix wird üblicherweise von links nach rechts und von oben nach unten gelesen. Diese Reihenfolge korrespondiert mit dem Koordinatensystems des Bildschirms, welcher seinen Ursprung links oben in der Ecke hat. Die Farbpunkte werden als Pixel (picture element) bezeichnet. Das Format, in der die Matrix von Farbpunkten und die notwendigen Metadaten wie z.B. Bildhöhe und Bildbreite gespeichert werden, wird als Bildformat bezeichnet. Es gibt eine grosse Anzahl solcher Bildformate, die sich in vielen Details unterscheiden können.

In dieser Übung verwenden Sie das Windows Bitmap Format zur Speicherung von Farbbildern. Pro Datei wird genau ein Bild unkomprimiert gespeichert. Im Ordner *Images* finden Sie drei Beispiele von Bilddateien: *rgb1.bmp* und *rgb2.bmp* sind im RGB-Format gespeichert und *argb.bmp* im ARGB-Format. Stellen Sie sicher, dass Sie über ein Programm verfügen, mit dem Sie die Bilder korrekt visualisieren können. Im Web finden Sie auch Online-Viewer für BMP.

RGB-Format: Wenn ein Bild im RGB-Format gespeichert ist, dann bedeutet dies, dass pro Bildpunkt drei Farbinformationen, nämlich Rot, Grün und Blau vorliegen. Diese Farbinformationen können vom Monitor mehr oder weniger direkt verwendet werden, um die farbigen RGB-Pixel des Monitors anzusteuern. In vielen Systemen benötigt ein RGB-Pixel je 1 Byte Speicherplatz pro Farbkanal, d.h. also 3 mal 1 Byte und somit 3 Byte pro Farbpixel. Ein unkomprimiertes Farbbild der Grösse 768 × 512 Pixel benötigt demnach mindestens $768 \cdot 512 \cdot 3 = 1'179'648$ Bytes Speicherplatz. Wenn Sie die Dateigrösse von *rgb1.bmp* anschauen, dann stellen Sie fest, dass die Dateigrösse leicht grösser ist, nämlich 1'179'702 Bytes. Die Differenz von 54 Bytes sind Metainformationen und werden als Bild-Header bezeichnet.

ARGB-Format: Das ARGB-Format ist eine Erweiterung des RGB-Formats in dem Sinne, dass pro Farbpixel nicht nur 3 Farbkanäle, sondern 4 Farbkanäle gespeichert werden können. Oft wird der vierte Kanal verwendet, um Transparenzinformationen pro Bildpunkt zu speichern. Wir nutzen diese Transparenzinformation hier nicht, sondern nutzen das ARGB-Format lediglich für den effizienteren Pixel-Zugriff, weil 4 Bytes pro Pixel besser in die Welt der 32-Bit- oder 64-Bit-Systeme passen, als 24 Bit pro Pixel.

2.1 Bild Header

Das Windows Bitmap Format unterstützt eine Vielzahl von Bildspeichervarianten, auf die wir hier aber nicht genauer eingehen wollen[1]. Je nach Bildspeichervariante werden mehr oder weniger Header-Informationen benötigt. Aus diesem Grund werden die Header-Informationen in unterschiedliche Headers gruppiert. Hier empfiehlt es sich, einen Blick auf die Wikipedia-Seite zum BMP Dateiformat zu werfen und sich einen ersten Überblick zu verschaffen. In dieser Übung interessieren uns nur der *Bitmap file header* (14 Bytes) und der *DIB header (bitmap information header)* der Grösse 40 Bytes.

2.1.1 Aufgabe 1

Definieren Sie in der Datei *Bitmap.h* die beiden öffentlichen Klassen `BitmapFileHeader` und `BitmapInfoHeader`. Listen Sie die Attribute der beiden Header in der korrekten Reihenfolge gemäss der Angaben auf der Wikipedia-Seite auf und verwenden Sie die korrekten Datentypen, z.B. `int32_t` oder `uint32_t`. Die `static_assert` Anweisung jeweils am Ende der beiden Klassendefinitionen stellt

[1] In den Modulen «Mathematik für Computergrafik und Bildverarbeitung» und «Bildverarbeitung» können Sie mehr zu diesem Thema lernen.

sicher, dass die beiden Header wirklich die korrekten Grössen 14 und 40 Bytes haben und somit im Total 54 Bytes Speicherplatz benötigen.

2.2 Klasse Bitmap

Die Klasse `Bitmap`, ebenfalls in der Datei *Bitmap.h* enthalten, definiert die notwendigen Daten zur Speicherung eines Bildes

- `m_fileHeader` vom Typ `BitmapFileHeader`
- `m_infoHeader` vom Typ `BitmapInfoHeader`
- `m_image` ein Zeiger zu einem Array vom Typ `uint8_t`

und listet die relevanten Methoden des abstrakten Datentyps `Bitmap` auf, z.B.:

- `read(…)`
- `write(…)`
- `invert()`

Da der logische Speicher einem eindimensionalen Array von Speicherzellen entspricht, speichern wir die Bilddaten in einem eindimensionalen Array auf dem Heap ab, obwohl die Bildmatrix zweidimensional ist. Das heisst, wir reihen Bildzeile an Bildzeile und transformieren das Bild somit von einer zweidimensionalen Matrix zu einem eindimensionalen Vektor. Als Elementdatentyp des Vektors verwenden wir `uint8_t`, weil jeder Bildpunkt je nach Format (RGB oder ARGB) entweder aus 3 oder 4 Bytes zusammengesetzt ist und somit `uint8_t` der grösste gemeinsame Teiler davon ist.

Der Zeiger zu den Bilddaten auf dem Heap könnte ein simpler Rohzeiger sein. Eleganter, moderner und sicherer ist es hingegen, einen `std::unique_ptr` zu verwenden, also einen Smart-Pointer, der selbständig dafür sorgt, dass die Daten auf dem Heap zur richtigen Zeit wieder freigegeben werden. Beachten Sie hier unbedingt, dass Sie `std::unique_ptr<uint8_t[]>` schreiben müssen, um anzugeben, dass auf dem Heap ein Array von `uint8_t` Werten verwaltet werden soll.

2.2.1 Aufgabe 2

Definieren Sie die notwendigen Attribute der Klasse `Bitmap` an der markierten Stelle. Verwenden Sie die zuvor angegebenen Attributnamen, um keine unnötigen Kompilationsfehler zu provozieren.

2.2.2 Aufgabe 3

Definieren Sie in der Datei *Bitmap.cpp* die beiden öffentlichen Klassen RGB und ARGB und implementieren Sie je einen benutzerdefinierten Konstruktor, um RGB nach ARGB und umgekehrt konvertieren zu können. Setzen Sie den Alpha-Wert von ARGB standardmässig auf 255.

2.2.3 Aufgabe 4

Implementieren Sie die folgenden Instanzmethoden

- `height()`: gibt die Bildhöhe in Anzahl Pixel, d.h. die Anzahl Bildzeilen, zurück; beachten Sie, dass das entsprechende Attribut im `BitmapInfoHeader` negativ sein kann. Wenn der Wert positiv ist, dann handelt es sich um ein Bottom-up-Bitmap, d.h. die unterste Bildzeile ist die erste im Speicher.
- `rowSize()`: gibt die Anzahl Bytes pro Bildzeile zurück; beachten Sie, dass der zurückgegebene Wert ein ganzzahliges Vielfaches von 4 sein muss; verwenden Sie nicht die Funktion `ceil()`, sondern berechnen Sie diesen Wert mit Ganzzahlarithmetik aus den entsprechenden Angaben im `BitmapInfoHeader`.
- `imageSize()`: berechnet den Speicherbedarf in Bytes für die Bilddaten mithilfe von height() und rowSize() und gibt ihn zurück. Verwenden Sie nicht das gleichnamige Attribut aus dem `BitmapInfoHeader`, weil dieses Attribut 0 sein darf.

2.2.4 Aufgabe 5

Für das Einlesen und später auch das Schreiben eines Bildes verwenden wir File-Stream-Klassen der Standardbibliothek. Da diese Klassen erst später im Semester behandelt werden, ist das Einlesen einer Bilddatei zu grossen Teilen bereits implementiert. Zwei Aspekte fehlen aber noch und müssen von Ihnen ergänzt werden:

- *Speicherallokation für die Bilddaten:* verwenden Sie `std::make_unique_for_overwrite` zur Allokation von Speicher auf einem Heap und speichern Sie den `unique_ptr` in der Instanzvariable `m_image` der Klasse `Bitmap`.
- *Bilddaten einlesen:* verwenden Sie die Methode `read(…)` der Klasse `std::ifstream` zum Einlesen der ganzen Bilddaten in einer einzigen Leseoperation. Im Fehlerfall soll eine

entsprechende Fehlermeldung auf die Konsole geschrieben und die Funktion mit dem Rückgabewert `false` beendet werden.

2.2.5 Aufgabe 6

Schreiben Sie die Bilddaten in die Bilddatei bzw. den File-Stream mit einem einzigen Aufruf von `write(…)` der Klasse `std::ofstream`. Sie dürfen beim Aufruf der Methode `Bitmap::write(…)` davon ausgehen, dass die beiden Bitmap-Header bereits abgefüllt sind. Im Fehlerfall soll eine entsprechende Fehlermeldung auf die Konsole geschrieben und die Funktion mit dem Rückgabewert `false` beendet werden.

2.3 Bildverarbeitung

Nachdem das Einlesen und Speichern von Bildern implementiert worden ist, kann der Fokus auf die Verarbeitung von Bildern gerichtet werden. Es sollen hier drei einfachere Bildverarbeitungsaufgaben realisiert werden.

2.3.1 Aufgabe 7

Implementieren Sie in der Instanzmethode `invert()` die farbliche Invertierung des Bildes. Die Methode soll kein neues Bild erstellen, sondern das vorhandene überschreiben. Da sich an der Geometrie des Bildes nichts ändern wird, müssen Sie die Header-Informationen nicht anpassen, sondern lediglich durch alle Bytes der Bilddaten iterieren und die einzelnen Bytes invertieren. Einen Byte-Wert *x* können Sie mit *~x* invertieren.

Stellen Sie sicher, dass diese Funktion mit beiden Formaten RGB und ARGB korrekt funktioniert.

2.3.2 Aufgabe 8

Implementieren Sie in der Instanzmethode `verFlip()` ein vertikales Spiegeln des Bildes. Vertikales Spiegeln wird durch umdrehen der Reihenfolge der Bildzeilen realisiert. Folgende Vorgehensweise empfiehlt sich:

- *iterieren Sie über die Hälfte aller Bildzeilen:* Verwenden Sie einen Rohzeiger `pRow1`, welcher auf das erste Byte der *i*-ten Bildzeile zeigt, einen zweiten Rohzeiger `pRow2`, welcher auf das erste Byte der (*h* – 1 – *i*)-ten Bildzeile zeigt, und nutzen Sie Zeigerarithmetik um `pRow1` und `pRow2` für die nächste Iteration vorzubereiten.
- *iterieren Sie über alle Bytes einer Bildzeile:* vertauschen Sie den entsprechenden Bytes von `pRow1` und `pRow2` mittels `std::swap(…)`.

Stellen Sie sicher, dass diese Funktion mit beiden Formaten RGB und ARGB korrekt funktioniert.

2.3.3 Aufgabe 9

Implementieren Sie in der Instanzmethode `horFlip()` ein horizontales Spiegeln des Bildes. Beachten Sie, dass Sie nicht einfach die Bytes jeder Bildzeile herumdrehen können, weil die Reihenfolge der Farbkanäle pro Bildpunkt nicht verändert werden darf. Stattdessen empfiehlt sich folgende Vorgehensweise:

- *iterieren Sie über alle Bildzeilen:* Verwenden Sie einen Rohzeiger `pRow`, welcher auf das erste Byte der aktuellen Bildzeile zeigt, und nutzen Sie Zeigerarithmetik um `pRow` für die nächste Iteration vorzubereiten.
- *reinterpretieren von* `pRow`*:* Verwenden Sie einen `reinterpret_cast`, um `pRow` in einen Rohzeiger `pPixels` umzuinterpretieren, welcher auf den Pixel der aktuellen Zeile zeigt.
- *Pixel einer Zeile vertauschen:* Iterieren Sie über die Hälfte der Pixel eine Bildzeile und vertauschen Sie jeweils die beiden Pixel `pPixel[i]` und `pPixel[wLast – i]` mittels `std::swap(…)`, wobei `wLast` der Index des letzten Pixels einer Bildzeile ist.

Stellen Sie sicher, dass diese Funktion mit beiden Formaten RGB und ARGB korrekt funktioniert.

2.4 Bildkonvertierungen

Zum Schluss sollen noch zwei Bildkonvertierungsfunktionen implementiert werden: `cvtToRGB()` und `cvtToARGB()`. Diese beiden Funktionen erstellen jeweils ein vollständig neues Bitmap, wobei die erstere aus einem RGB-Bitmap ein ARGB-Bitmap erstellt und die zweitere umgekehrt.

2.4.1 Aufgabe 10

Beachten Sie bei der Implementierung der beiden Funktionen, dass im neu erzeugten Bitmap `out` alle drei Attribute korrekt initialisiert werden müssen. Den File-Header können Sie beispielweise ganz einfach durch `out.m_fileHeader = m_fileHeader` initialisieren, da die entsprechende Klasse über einen

automatisch generierten Zuweisungsoperator verfügt. Beim Info-Header können Sie analog vorgehen, müssen aber bedenken, dass sowohl `m_bpp` als auch `m_imageSize` entsprechend dem neuen Pixel-Format angepasst werden müssen. Nachdem die beiden Header richtig initialisiert worden sind, können die neuen Bilddaten auf dem Heap alloziert und korrekt gesetzt werden.

3 Abgabe

Senden Sie die drei folgenden Dateien ungezippt als E-Mail-Anhänge an christoph.stamm@fhnw.ch:

1. «Bitmap.h» mit den von Ihnen implementierten Ergänzungen in der Klasse `Bitmap`. Schreiben Sie Ihren vollständigen Namen als Kommentar an den Anfang der Datei. Dabei bezeugen Sie, dass Sie die Implementierung eigenständig erstellt haben.
2. «Bitmap.cpp» mit den von Ihnen implementierten Ergänzungen. Schreiben Sie Ihren vollständigen Namen als Kommentar an den Anfang der Datei. Dabei bezeugen Sie, dass Sie die Implementierung eigenständig erstellt haben.
3. Erstellen Sie einen Screenshot von allen ausgeführten Unit-Tests.