FDS2 Medizin- und Bioinformatik SS 2025, Übung 2

Name: _____ **Aufwand in h:** _____

Punkte: _____ **Kurzzeichen Tutor/in:** _____

Beispiel 1 (100 Punkte): Bitmaps

Implementieren Sie im Namensraum **`graphics`** einen ADT **`bitmap`**, mit dem 24-Bit-Bitmaps manipuliert werden können. Dieser ADT ist als `struct` zu implementieren und muss in der Schnittstelle die im Folgenden beschriebenen Methoden anbieten.

Schnittstellenfunktionen zum Erzeugen von Objekten des Typs bitmap:

```
graphics::bitmap * generate_bitmap ();
graphics::bitmap * generate_bitmap (std::size_t const w, std::size_t const h, graphics::pixel_type const c = graphics::white);
graphics::bitmap * generate_bitmap (char const * const p_name);
graphics::bitmap * generate_bitmap (std::string const & name);
graphics::bitmap * generate_bitmap (std::istream & in);
graphics::bitmap * generate_bitmap (bitmap const & src);
```

`generate_bitmap()` erzeugt ein Bitmap der Größe 0*0 Pixel. Mit z. B. den Funktionen `resize` oder `read_from` kann dessen Größe nachträglich verändert werden. Die Parameter *w* (width) und *h* (height) geben die Größe des zu konstruierenden Bitmaps (in Pixels) an. Das mit *w* und *h* konstruierte Bitmap erhält die Farbe *c*.

Schnittstellenfunktionen zum Initialisieren von Objekten des Typs bitmap:

```
void clear (graphics::bitmap & bmp);
void resize (graphics::bitmap & bmp, std::size_t const w, std::size_t const h, graphics::pixel_type const c = graphics::white);
```

Die Methode `clear` ist ein Synonym für den Methodenaufruf `resize(bmp,0,0)`. Mit der Methode `resize` kann die Größe eines Bitmaps verändert werden. Die Parameter *w* und *h* geben die neue Größe (in Pixel) eines Bitmaps an. Das so veränderte Bitmap erhält die Farbe *c*. (Auch dann, wenn die neuen Dimensionen gleich den alten sind.)

Funktion zum Vergleichen von Objekten des Typs `bitmap`:

```
bool equals (graphics::bitmap const & lhs, graphics::bitmap const & rhs);
```

Schnittstellenfunktionen für den Attributzugriff auf Objekte des Typs bitmap:

```
graphics::pixel_type & at (graphics::bitmap & bmp, std::size_t const x, std::size_t const y);
graphics::pixel_type const & at (graphics::bitmap const & bmp, std::size_t const x, std::size_t const y);
graphics::long_type get_height (graphics::bitmap const & bmp);
graphics::long_type get_width (graphics::bitmap const & bmp);
graphics::long_type get_image_size (graphics::bitmap const & bmp);
graphics::long_type get_num_pixels (graphics::bitmap const & bmp);
graphics::byte_type * get_image (graphics::bitmap & bmp);
graphics::byte_type const * get_image (graphics::bitmap const & bmp);
graphics::pixel_type * get_pixels (graphics::bitmap & bmp);
graphics::pixel_type const * get_pixels (graphics::bitmap const & bmp);
```

Die Methode `at` liefert eine Referenz auf das mit `x` und `y` spezifizierte Pixel. Die Methode `get_image_size` liefert die Anzahl der Bytes, die die geladenen Bilddaten im Speicher benötigen. Die Methoden `get_image` bzw. `get_pixels` liefern Zeiger auf die Bilddaten (siehe dazu die Konzeptbilder). Es ist garantiert, dass via `get_image` bzw. `get_pixels` auf mindestens `get_image_size` Bytes bzw. `get_num_pixels` Pixels zugegriffen werden kann.

Schnittstellenfunktionen für das Lesen und Schreiben von Objekten des Typs `bitmap`:

```
bool read_from (graphics::bitmap & bmp, char const * const p_name);
bool read_from (graphics::bitmap & bmp, std::string const & name);
bool read_from (graphics::bitmap & bmp, std::istream & in);
bool write_to (graphics::bitmap const & bmp, char const * const p_name);
bool write_to (graphics::bitmap const & bmp, std::string const & name);
bool write_to (graphics::bitmap const & bmp, std::ostream & out);
```

Diese Methoden lesen bzw. schreiben ein Bitmap. Es können dabei Dateinamen (Parameter `p_name` und `name`) oder Dateiströme (Parameter `in` und `out`) angegeben werden. Dateinamen müssen die Erweiterung `bmp` besitzen. Die Funktionen liefern dann true, wenn alle Lese- bzw. Schreiboperationen erfolgreich durchgeführt werden konnten.

Schnittstellenfunktionen für das Manipulieren von Bilddaten von Objekten des Typs `bitmap`:

```
void detect_edges (graphics::bitmap & bmp);
void fill (graphics::bitmap & bmp, graphics::pixel_type const color = graphics::white);
void invert (graphics::bitmap & bmp);
void to_gray (graphics::bitmap & bmp);
```

Die Methode `detect_edges` wendet einen Kantendetektionsalgorithmus auf ein geladenes Bitmap an. Verwendet werden dabei die Sobel-Operatoren S_x und S_y, die mit $\sqrt{{S_x}^2 + {S_y}^2}$ verknüpft werden (siehe dazu auch http://en.wikipedia.org/wiki/Sobel_operator):

$$S_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \qquad S_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$

Die Methode `fill` füllt ein Bitmap mit der gegebenen Farbe. Die Methode `invert` invertiert die Farben eines Bitmaps. Die Methode wandelt die Farben eines Bitmaps in die entsprechenden Graustufen um.

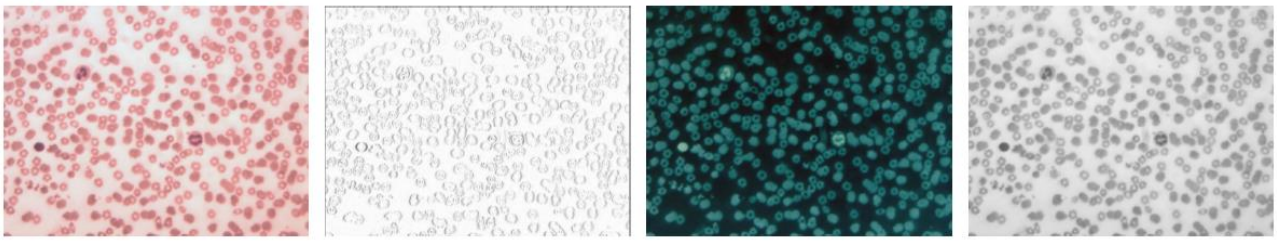

Abb. 1: Originalbild; nach der Kantendetektierung; nach dem Invertieren; nach der Umwandlung in Graustufen

Weitere Schnittstellenfunktionen für Objekte des Typs `bitmap`:

```
void swap (graphics::bitmap & lhs, graphics::bitmap & rhs);
```

Die Funktion `swap` vertauscht die Inhalte der beiden beteiligten Bitmaps.

Eine Beispielanwendung könnte wie folgt aussehen:

```
#include "./bitmap.h"
#include <cassert>
void main () {
  graphics::bitmap b0;
  graphics::generate_bitmap(b0, 300, 100);
  graphics::bitmap b1;
  graphics::generate_bitmap(b1, 300, 100, graphics::red);
  graphics::bitmap b2;
  graphics::bitmap b3;

  b2 = b1; assert (graphics::equals(b2,b1));

  graphics::read_from (b3, "./data/erythrocytes.bmp");
  graphics::to_gray (b3);
  graphics::detect_edges (b3);
  graphics::invert (b3);

  graphics::swap (b2, b3);

  graphics::write_to (b1, "./data/output-1.bmp");
  graphics::write_to (b2, "./data/output-2.bmp");
  graphics::write_to (b3, "./data/output-3.bmp");
}
```

Die folgenden Konzeptbilder sowie die Datei „BMP File Format.pdf“ erläutern notwendige Details über den Aufbau von bmp-Dateien. Siehe dazu auch die Vorlesung bzw. die Übung.

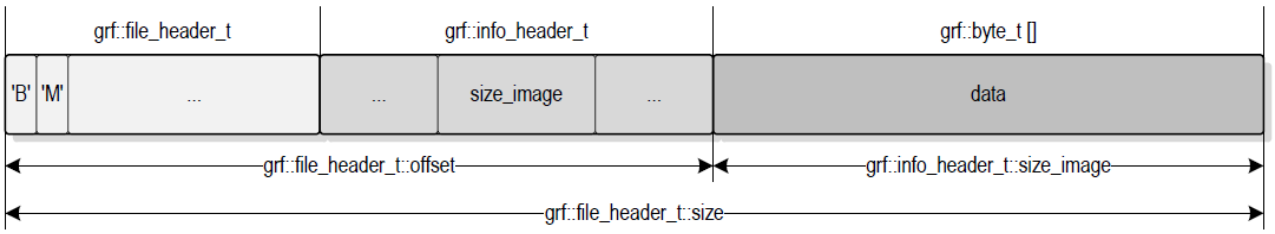

Abb. 2: Der Dateiaufbau von 24-Bit-Bitmaps

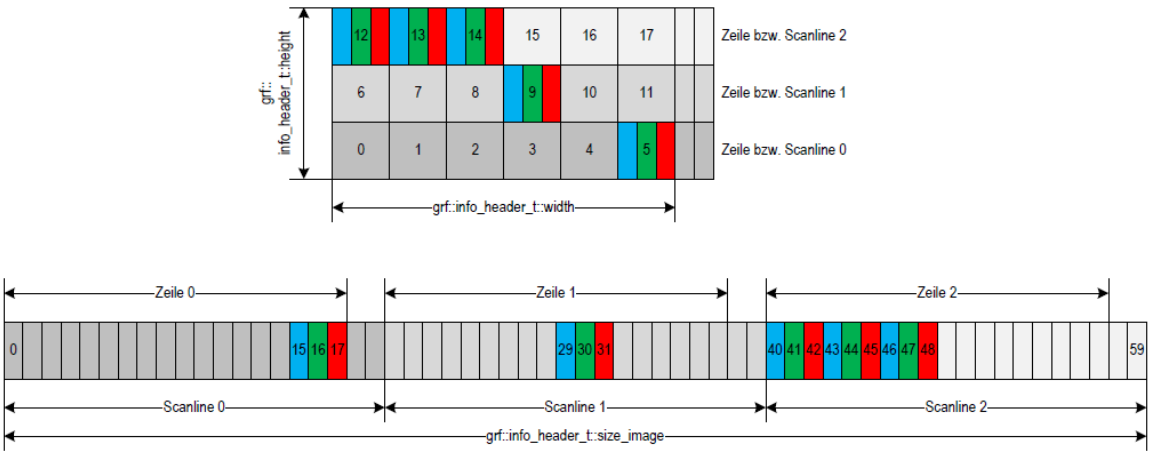

Abb. 3: Der Bildaufbau von 24-Bit-Bitmaps

Für die Implementierung des ADTs `bitmap` verwenden Sie die folgenden Typdefinitionen (alle im Namensraum `graphics`):

```
#include <cstdint>

typedef uint_fast8_t byte_type;
typedef uint_fast32_t uint32_type;
typedef int_fast32_t long_type;
typedef uint16_t uint16_type;

#pragma pack (push, 1)
struct pixel_type {
  byte_type blue;
  byte_type green;
  byte_type red;
};
#pragma pack (pop)

#pragma pack (push, 1)
struct file_header_type {
  union {
    byte_type signature [2]; // file type; must be 'BM'
    uint16_type type; // file type; must be 0x4d42
  };
  uint32_type size; // size, in bytes, of the bitmap file
  uint16_type reserved_1; // reserved; must be 0
  uint16_type reserved_2; // reserved; must be 0
  uint32_type offset; // offset, in bytes, from the beginning of the 'file_header_t' to the bitmap
bits
};
#pragma pack (pop)

#pragma pack (push, 1)
struct info_header_type {
  uint32_type size; // number of bytes required by the structure
  long_type width; // width of the bitmap, in pixels
  long_type height; // height of the bitmap, in pixels
  uint16_type planes; // number of planes for the target device; must be 1
  uint16_type bit_count; // number of bits per pixel
  uint32_type compression; // type of compression; 0 for uncompressed RGB
  uint32_type size_image; // size, in bytes, of the image
  long_type x_pels_pm; // horizontal resolution, in pixels per meter
  long_type y_pels_pm; // vertical resolution, in pixels per meter
  uint32_type clr_used; // number of color indices in the color table
  uint32_type clr_important; // number of color indices that are considered important
};
#pragma pack (pop)
```

Anmerkungen: (1) Geben Sie Lösungsideen an. (2) Strukturieren und arbeiten Sie sauber. (3) Kommentieren Sie ausführlich. (4) Geben Sie ausreichend Testfälle ab. (5) Prüfen Sie alle Eingabedaten auf ihre Gültigkeit.