Hochschule Karlsruhe
University of Applied Sciences

Fakultät für **Informatik**
und Wirtschaftsinformatik

Prof. Dr. Carsten Sinz
Lehrgebiet Software-Entwicklung

Labor Betriebssysteme: Aufgabe 2
Dateien

Hinweise:

- Bearbeiten Sie die Teilaufgaben in Teams und laden Sie Ihre Antworten in ILIAS hoch. Zusätzlich müssen Sie Ihre Abgaben auch im Labor präsentieren, dabei müssen alle Teammitglieder anwesend sein.
- Für Teilaufgaben 1 und 2 sind nur textuelle Abgaben gefordert. Für Teilaufgabe 3 geben Sie Ihr gesamtes C-Projekt ab, d. h. alle Quelldateien und alles was dazugehört, damit wir Ihr Projekt mittels `make` bauen können.
- Für Teilaufgabe 3 gibt es Beispieldateien im Ilias im Archiv `example.zip`. Dort ist eine C-Quelldatei enthalten, die den Einstieg erleichtern soll, und zwei Bitmaps. Eine kleine Bitmap zum Ausprobieren und eine größere Bitmap-Datei. Das Verarbeiten der größeren Bitmap-Datei werden Sie bei der Abgabe live vorführen.

In diesem Übungsblatt werden Sie ein Programm schreiben, dass eine **Bitmap-Bilddatei** *faltet*. Dafür müssen Sie zuerst verstehen, wie Bitmap-Bilddateien aufgebaut sind und welche **Eigenheiten** diese aufweisen, sodass Sie am Schluss wieder eine korrekte **Bitmap-Bilddatei** erstellen können. Dazu werden Sie **System Calls** wie **open()**, **read()**, **write()** und **lseek()** verwenden.

Teilaufgabe 1 (Header inspizieren):

Wir wollen in diesem Teil den Aufbau von **Bitmap-Bilddateien** näher betrachten. Mit Bitmap-Bilddateien (nachfolgend nur noch **Bitmap** genannt) sind hierbei Bilddateien gemeint, die üblicherweise die Endung `.bmp` besitzen. **Bitmaps** haben **zwei Headerbereiche** und **einen Bildbereich**. Eventuell kann **vor dem Bildbereich** noch eine **Farbpalette** definiert sein.

Der **erste Headerbereich** wird *bitmap file header* (nachfolgend nur *File-Header*) genannt, während der **zweite Headerbereich** *DIB header* (oder bitmap information header, nachfolgend nur *Info-Header*) genannt wird. Alle Felder sind Ganzzahlen und **unsigned**, sofern nicht mit **signed** markiert.

Der **File-Header** umfasst 14 Bytes mit den folgenden Feldern:

- **Signatur** (Eine Bitmap hat hier die beiden Bytes 0x42 0x4D, welche im ASCII-Zeichensatz für B M stehen) (2 Bytes)
- **Dateigröße** (4 Bytes)
- Reservierte Bytes (Sind nicht näher spezifiziert und können beliebig belegt werden) (4 Bytes)
- **Daten-Offset** (Gibt den Versatz an, an dem die eigentlichen Bilddaten anfangen) (4 Bytes)

Es gibt mehrere Varianten bzw. Versionen des **Info-Headers**, weshalb wir uns hier auf eine Variante beschränken, die am weitesten verbreitet ist.

Die Variante, die hier behandelt wird, ist als `BITMAPINFOHEADER` bekannt und umfasst 40 Bytes. Nachfolgend eine Auflistung der darin enthaltenden Felder:

- **Header-Größe** (in diesem Fall sollte hier immer die Zahl 40 stehen) (4 Bytes)
- **Bildbreite** in Pixel (4 Bytes, **signed**)
- **Bildhöhe** in Pixel (4 Bytes, **signed**)
- Anzahl der Farbebenen (2 Bytes)
- **Bits pro Pixel** (2 Bytes)
- Kompressionsmethode (4 Bytes)
- **Bildgröße** (Größe des Bildbereichs in Bytes) (4 Bytes)
- Horizontale Auflösung (DPI oder PPM) (4 Bytes, **signed**)
- Vertikale Auflösung (DPI oder PPM) (4 Bytes, **signed**)
- Anzahl der Farben in der Palette (4 Bytes)
- Anzahl der wichtigen Farben (4 Bytes)

Diese **Headerfelder** sollen Sie nun in **Bitmap-Dateien** ausfindig zu machen. Im Ilias finden Sie eine **ZIP-Datei** mit dem Namen `inspect.zip`, die fünf Bitmap-Dateien beinhaltet. Schauen Sie sich die Dateien in einem **Hex-Editor** Ihrer Wahl an (z. B. hexed.it) und bearbeiten Sie die Aufgaben.

a) Schauen Sie sich das Feld für die **Dateigröße** der Datei `A.bmp` im *File-Header* an und ermitteln Sie daraus die **Byte-Reihenfolge** (*endianness*).

b) Geben Sie alle Werte für den `BITMAPINFOHEADER` der Datei `A.bmp` an.

c) Vergleichen Sie die Werte für **Bildbreite** und **Bildhöhe** der Dateien `C_1.bmp` und `C_2.bmp` und stellen Sie eine Hypothese auf, was dies bedeuten könnte.

d) Vergleichen Sie den `BITMAPINFOHEADER` von `C_3.bmp` mit dem, der anderen Dateien. Was fällt Ihnen auf?

Teilaufgabe 2 (Pixeldaten inspizieren): Nun werden die **Pixeldaten** genauer inspiziert. Benutzen Sie weiterhin Ihren **Hex-Editor** und bearbeiten Sie die nachfolgenden Aufgaben.

a) Schauen Sie sich die **Pixeldaten** von `A.bmp` und `B.bmp` an und machen Sie sich bewusst, welche Werte für die **Bildbreite** und **Bildhöhe** festgelegt sind. Anhand dieser Dimensionen, versuchen Sie nun zu verstehen, wie die blauen, grünen und roten Pixel jeweils abgespeichert werden. Was fällt Ihnen auf?

   Hinweis: `A.bmp` hat vier Reihen blaue Pixel, `B.bmp` hingegen nur drei.

b) Versuchen Sie nachfolgend die **Pixeldaten** in `C_1.bmp` und `C_2.bmp` zu interpretieren. Erklären Sie, warum der Computer das Bild in beiden Fällen gleich darstellen kann und welche grundlegende Idee dahintersteckt.

c) Die Datei `C_3.bmp` ist wesentlich kleiner als `C_1.bmp` und `C_2.bmp`, obwohl der gleiche Bildinhalt dargestellt wird. Wie wird in `C_3.bmp` der Bildinhalt komprimiert und anhand von welchen Werten könnte der Computer dies feststellen? Sie müssen vollständig erklären, wie das Bild rekonstruiert werden kann!

   Hinweis: Vergleichen Sie den Wert des **Daten-Offsets** mit den anderen Dateien.

Vorbereitung Teilaufgabe 3: Datei-Operationen

Öffnen und Schließen

Zur **Dateiverarbeitung** muss eine Datei zuerst geöffnet werden. Dabei werden Existenz, Berechtigungen und Dateipfad geprüft. In C wird die `open()` Funktion aus der C-Bibliothek (libc), welche einen Wrapper für den gleichnamigen System-Call bereitstellt, verwendet, um eine Datei zu öffnen. Der Befehl `man 2 open` hilft bei der Nutzung und findet die deklarierenden Header-Dateien. `open()` erfordert den **Dateipfad** (`pathname`) und den Parameter `flags`. Gebräuchliche Werte für `flags` ermöglichen **Lese**- oder **Schreibzugriffe**, z. B.:

Wert	Bedeutung
O_EXCL	Datei wird vom Programm exklusiv geöffnet
O_RDONLY	Datei wird nur zum Lesen geöffnet
O_WRONLY	Datei wird nur zum Schreiben geöffnet
O_RDWR	Datei wird zum Lesen und Schreiben geöffnet
O_CREAT	Datei wird neu angelegt
O_TRUNC	Inhalt der Datei wird beim Öffnen gelöscht

Weitere `flags`-Werte finden sich auf der `open()`-Hilfsseite.

Mehrere `flags`-Werte können mit einem logischen *Oder* | verknüpft werden, z. B. `O_CREAT | O_RDWR`.

Beispiel für `open()`:

```
int fd = open("1.txt", O_CREAT | O_RDWR, 0666);
```

`open()` gibt einen **Integer-Wert** (Dateideskriptor) zurück, der später zum Zugriff auf die Datei verwendet wird. Der dritte Parameter im Beispiel gibt die **Zugriffsrechte** an, die gesetzt werden sollen, wenn die Datei erstellt wird.

> **Hinweis**: Zahlen mit vorangestellter 0 (im Beispiel oben 0666) werden, wie z. B. auch in Java, als Oktalzahlen interpretiert.

Die Methode `close()` schließt Dateien und kann mit `man 2 close` nachgeschlagen werden. `close()` erhält als Argument den **Dateideskriptor** `fd`.

Fehler beim Öffnen und Schließen von Dateien (z. B. nicht existente Datei) führen zu einem Rückgabewert von -1 bei `open()` und `close()`. Fehlerverhalten kann auf **Hilfeseiten** nachgelesen werden.

Fehlercodes

Im Fehlerfall speichern **POSIX-konforme Funktionen** (wie z. B. `open()`, `close()`, `read()`, `write()` und `lseek()`) der C-Bibliothek den Fehlercode in der globalen Variable **errno**. Diese Variable ist aussagekräftig, wenn eine Methode der C-Bibliothek, die den POSIX-Anforderungen entspricht, einen Fehler signalisiert. Bei einem Aufruf von `open()` mit Rückgabewert -1 kann `errno` zur Fehlerbestimmung überprüft werden:

```
int fd = open("1.txt", O_RDONLY);
if (fd == -1) {
  printf("Error: Could not open file, code %n", errno);
  exit(EXIT_FAILURE);
}
```

`exit()` beendet die Programmausführung und liefert der Shell den Fehlercode `EXIT_FAILURE`.

Mögliche `errno`-Werte sind in `errno.h` definiert und können mit `man errno` nachgelesen werden. Mit definierten Konstanten kann `errno` (z. B. mit `ENOENT`) verglichen werden, um **Fehlerursachen** zu identifizieren:

```
if (fd == -1) {
  if (errno == ENOENT) {
    printf("Error: File does not exist");
  }
  else {
    printf("Error: Could not open file, code %n", errno);
  }
  exit(EXIT_FAILURE);
}
```

Lesen von Dateien

Die Methode `read()` ermöglicht das Lesen aus einer geöffneten Datei (siehe Hilfsseite), wobei der **Dateideskriptor** die Datei angibt. `read()` liest `count` Bytes und speichert sie im Speicherbereich, auf den `buf` zeigt, welcher mindestens `count` Bytes groß sein sollte. `count` ist vom Typ `size_t`.

Die **Lese-Position** ist implizit durch den Dateideskriptor bestimmt und ändert sich durch `read()`-Operationen oder explizit durch `lseek()`. Nach dem Öffnen ist die Position 0 (Dateianfang).

`read()` gibt die Anzahl der gelesenen Bytes zurück, wenn erfolgreich, und verschiebt die Dateiposition entsprechend. Der Wert kann kleiner als `count` sein, wenn das Dateiende erreicht ist.

Lesebeispiel aus einer Datei:

```
int count = 10;
char *buf = (char *) malloc((count + 1) * sizeof(char));
int ret = read(fd, buf, count);
buf[ret] = '\0';
printf("%s", buf);
```

> **Wichtig**: In C muss ein String mit einem 0-Byte enden, während die von `read()` gelesenen Bytes meistens nicht damit enden. Daher wird das letzte Byte explizit auf 0 gesetzt!

Die Fehlerbehandlung für `read()` ist analog zur oben genannten von `open()` und kann in den Hilfe-Seiten nachgeschlagen werden.

Schreiben von Dateien

Ähnlich wie `read()` existiert die Methode `write()`, welche in den entsprechenden Hilfe-Seiten definiert ist.

Anwendungsbeispiel:

```
const char *buf = "Hello world!\n";
int ret = write(fd, buf, strlen(buf));
```

`strlen()` bestimmt dabei die Länge der Zeichenkette.

Lesen und Schreiben von Structs

Bisher haben wir `read()` und `write()` für Arrays vom Typ `char` verwendet. Der übergebene Zeiger hat jedoch den Typ `void*`, wodurch er auf beliebige Speicherbereiche und somit auch auf komplexe Datenstrukturen verweisen kann. Zum Beispiel können Strukturen auf diese Weise geschrieben werden (Auf Fehlerbehandlungen wurde zwecks besserer Übersichtlichkeit verzichtet):

```c
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <fcntl.h>

typedef struct {
    int id;
    char name[100];
    float salary;
} Employee;

int main() {
    Employee emp1, emp2;

    emp1.id = 1;
    strcpy(emp1.name, "Alice");
    emp1.salary = 50000.0;

    // Create and open a file
    int fd = open("employee.bin", O_CREAT | O_RDWR, 0644);

    // Write the struct to the file
    write(fd, &emp1, sizeof(Employee));

    // Use lseek to move the file position back to the start
    lseek(fd, 0, SEEK_SET);

    // Read the struct from the file
    read(fd, &emp2, sizeof(Employee));

    close(fd);

```

```
    // Print the data read from the file
    printf("ID: %d\nName: %s\nSalary: %f\n", emp2.id, emp2.name, emp2.salary);

    return 0;
}
```

Padding-Bytes in Structs

Das Schreiben und Lesen eigener `Structs` ist grundsätzlich unproblematisch, jedoch können **Padding-Bytes** bei der Verwendung von `sizeof` zur Größenprüfung auffallen. Beim **File-Header** führt dies zu einer Größe von **16** anstatt **14**. Dies liegt am standardmäßigen **Byte-Alignment**, welches anhand des größten Datentyps ein **Byte-Boundary** festlegt (hier 4 Bytes, z. B. `u_int32_t`). Da das Signaturfeld nur 2 Bytes groß ist, müssen 2 zusätzliche Bytes hinzugefügt werden, um das Byte-Boundary einzuhalten. Um das Byte-Alignment zu beeinflussen, können **Pragma-Directives** als spezielle Compiler-Anweisungen genutzt werden:

```
#include <stdio.h>
#include <stdint.h>

typedef struct {
    uint16_t signature;
    uint32_t fileSize;
    uint32_t reserved;
    uint32_t dataOffset;
} UnpackedFileHeader;

#pragma pack(push, 1)
typedef struct {
    uint16_t signature;
    uint32_t fileSize;
    uint32_t reserved;
    uint32_t dataOffset;
} PackedFileHeader;
#pragma pack(pop)

int main()
{
    // Packed should be 14 and Unpacked should be 16
    printf("Packed: %lu, Unpacked: %lu",
      sizeof(PackedFileHeader),
      sizeof(UnpackedFileHeader)
    );
    return 0;
}
```

`pack(push, 1)` verändert das Byte-Alignment so, dass der Struct auf einer 1-Byte-Boundary liegen wird. Dadurch werden keine Padding-Bytes mehr benötigt und der File-Header kann ohne Probleme in diese Datenstruktur geladen werden.

Teilaufgabe 3 (3×3-Filter): Jetzt sollen die vorher erarbeiteten Informationen zum Tragen kommen. Ihre Aufgabe wird sein, ein Programm in **C** zu schreiben, das einen Filter auf ein bestehendes Bild anwendet. Sie dürfen keine externen Libraries einbinden, Sie dürfen aber gerne selbst Strukturen erstellen, mit denen Sie die Bilddaten einfacher handhaben können. Nachfolgend beschreiben die einzelnen Aufgaben die weiteren Anforderungen an dieses Programm.

a) Ihr Programm soll einen *Dateinamen* und einen Namen für die geforderte Filter-Operation annehmen. Vordefinierte Filter sind auf der nächsten Seite zu finden.

b) Überprüfen Sie, ob die Datei sich öffnen lässt und ob die ersten beiden Bytes der Signatur entsprechen. Geben Sie bei Fehlern entsprechende Meldungen zurück.

c) Verarbeiten Sie nur Bitmaps mit 24 Bit pro Pixel, ohne Farbpalette und ohne Komprimierung.

d) Stellen Sie sicher, dass das Bild mindestens drei Pixel breit und mindestens drei Pixel hoch ist.

e) Wenn Sie den Filter auf die Bilddaten anwenden, verwerfen Sie dabei Randpixel, d.h. die Dimensionen reduzieren sich um jeweils 2 Pixel. Zum Beispiel, ein Bild mit Breite 3 und Höhe 3, hat danach lediglich die Breite 1 und die Höhe 1.

f) Mit den veränderten Bilddaten erstellen Sie eine neue Datei. Denken Sie daran die Headerinformationen entsprechend zu aktualisieren und Erkenntnisse aus den ersten beiden Aufgaben zu berücksichtigen.

Diskrete 2D-Faltung

In der Bildverarbeitung werden oft Filteroperationen durchgeführt. Diese Filter können unterschiedliche Effekte hervorrufen, z.B. Schwarz-Weiß-Bilder, Schärfe, Unschärfe oder Konturen erkennen. Bei Filtern, die Informationen aus mehr als nur einem einzigen Pixel zusammenführen, sind Randpixel problematisch, da dem entsprechenden Filter Bildinformationen fehlen, um den Filter korrekt anzuwenden. Es gibt mehrere Strategien, um dieses Randpixel-Problem anzugehen, hier beschränken wir uns auf die Strategie *Crop* bzw. *Avoid Overlap*, die dazu führt, dass das resultierende Bild geringere Dimensionen aufweisen wird.

Eine Faltung beschreibt hier das Anwenden eines Filters (hier eine 3×3-Matrix; auch Kernel genannt) auf ein bestehendes Bild, dabei wird keine traditionelle Matrix-Multiplikation angewendet.

Jeder Pixel des resultierenden Bildes ist die gewichtete Summe der Pixel des Ausgangsbildes. Die Auswahl der Ausgangspixel wird durch den gewählten Kernel beschränkt.

Wie eine Faltung eines solchen Kernels durchgeführt wird, können wir mittels einer Formel beschreiben. Koordinaten sind als (x,y) zu lesen, d.h. x beschreibt die horizontale (nach rechts positiv) und y die vertikale (nach unten positiv) Achse. Dabei definieren wir I als Eingabebild mit (0,0) als Pixel links oben, Kernel K in Form einer 3×3-Matrix und O als das Ausgabebild. Die Mitte des Kernels wird mit (0,0) definiert. Beschreibt b die Breite von I und h die Höhe von I, so gilt $i \in \{1, 2, 3, \ldots, b-1\}$ und $j \in \{1, 2, 3, \ldots, h-1\}$. So ergibt sich für einen beliebigen Bildpunkt im Ausgabebild O folgende Formel:

$$O_{i,j} = \sum_{p=-1}^{1} \sum_{q=-1}^{1} I_{i-p,j-q} \cdot K_{p,q}$$

Vordefinierte Filter

$$smooth = \frac{1}{9}\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

$$sharp = \begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}$$

$$edge = \begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}$$

$$emboss = \begin{bmatrix} 2 & 1 & 0 \\ 1 & 1 & -1 \\ 0 & -1 & -2 \end{bmatrix}$$

Zur Visualisierung der Filter, also damit Sie wissen, ob Ihr Filter auch das richtige macht, können Sie z.B. folgende Webseite verwenden:

https://setosa.io/ev/image-kernels/

Auf der Webseite können Sie dann weiter unten beliebige Filter mit eigenen Bildern testen.

Hinweis: Das Beispiel auf der Webseite rechnet etwas anders, als die Formel, die auf diesem Blatt angegeben ist. Auf der Webseite wird der Wert *links oben* im Kernel auch mit dem Wert *links oben* des Bildes verrechnet. Schauen Sie sich die auf diesem Blatt gegebene Summenformel genauer an, werden Sie merken, dass zuerst *links oben* im Bildbereich mit dem Wert des Kernels *rechts unten* verrechnet wird.