

1 Ein- und Ausgabe

Aufgabe 1.1. (Kopierprogramm – leicht) Schreiben Sie ein Programm, mit dem Sie eine Datei in eine andere kopieren können. Der Nutzer soll in der Lage sein, das Programm wie folgt aufzurufen:

```
./kopiere Quelle Ziel
```

Folgende Funktionen sind zur Lösung der Aufgabe hilfreich: `fopen(3)`, `fread(3)`, `fwrite(3)`, und `fclose(3)`.

Implementieren Sie den Kopiervorgang, indem Sie wiederholt in einer Schleife ein Stück der Eingabedatei in einen Puffer einlesen und dieses in die Ausgabedatei schreiben. Lesen Sie die Dokumentation von `fread(3)`, spezifisch den Abschnitt *Return Value*, um herauszufinden, wie Sie das Ende der Eingabedatei erkennen.

Experimentieren Sie mit verschiedenen Puffergrößen. Können Sie einen Unterschied in der Geschwindigkeit feststellen?

Aufgabe 1.2. (PPM-Bilder lesen / schreiben – leicht) PPM (portable pixmap) ist eine Familie einfacher Bildformate. Recherchieren Sie im Internet, wie PPM-Bildformate funktionieren und schreiben Sie eine Funktion, die ein PPM-Bild einliest, und als Datenstruktur im Computer ablegt. Sie brauchen nur PPM-Bilder vom Typ P3 zu unterstützen.

Das Bild soll durch folgende Struktur dargestellt werden:

```
1 /* ein einzelnes Pixel */
2 struct pixel {
3     int rot, weiss, gruen;
4 };
5
6 /* ein PPM Typ P3 Bild */
7 struct ppm_bild {
8     /* Kopfdaten */
9     int breite, hoehe;
10    int max_helligkeit;
11
12    /* mit malloc() allozieren */
13    struct pixel *leinwand;
14};
```

Anschließend schreiben Sie eine Funktion, die eine `ppm_bild` Struktur als Typ P3 PPM-Bild in eine Datei schreibt.

Prüfen Sie die Korrektheit Ihrer Funktion, indem Sie diese in einem Programm verwenden, welches ein PPM-Bild einliest und wieder in eine Datei schreibt. Das entstehende Bild muss identisch zum Original sein.

Aufgabe 1.3. (PPM-Bildverarbeitung – mittel/schwer) Schreiben Sie ein einfaches Bildverarbeitungsprogramm, welches auf Typ P3 Bildern arbeitet. Das Bildverarbeitungsprogramm soll in der Lage sein, einfache Transformationen auf dem Bild auszuführen. Hier sind ein paar Beispiele nach aufsteigender Schwierigkeit sortiert:

- Konvertierung von Farbe nach Graustufen oder Schwarzweiß

- Zuschneiden des Bildes
- Drehung um 90° , 180° , und 270° , sowie Spiegelung
- Filterung mit Faltungsmatrizen (Gaußfilter, Schärfungfilter, Kantenfilter, etc.)
- Skalierung
- Drehung um beliebige Winkel

Sie können sich auch gerne eigene Operationen ausdenken.

Aufgabe 1.4. (Termauswerter – schwer) Schreiben Sie ein Programm, in das Sie einen C-Ausdruck mit Variablen eingeben können. Das Programm soll den Nutzer nach Belegungen für die Variablen fragen und den Wert des Terms ermitteln. Verwenden Sie für alle Variablen den Typ `long long int`. Sie können zur Lösung der Aufgabe den *Shunting-Yard-Algorithmus* verwenden.

Aufgabe 1.5. (tar(1) – schwer) In dieser Aufgabe sollen Sie eine vereinfachte Version des UNIX-Archivprogramms `tar` implementieren, um Archive im Format „ustar“ (UNIX Standard Tape ARchiver) zu erstellen. Ein Archiv ist eine Datei, welche die Inhalte von anderen Dateien enthält, nebst Meta-Informationen über eben diese – archivierten – Dateien (d. h. Dateiname, zuletzt modifiziert, ...).

Archivformat Ein TAR-Archiv besteht aus Blöcken von 512 Bit-Oktets (für uns: Bytes) und schreibt schlicht die Inhalte der Dateien – in solche Blöcke unterteilt – hintereinander. Vor jedem Dateinhalt steht noch ein Header der die Meta-Informationen bereithält, welchen wir als Datenstruktur in der Datei `ti3tar.h` für euch mitliefern. Bei uns werden alle Daten in ASCII encodiert, das bedeutet, dass auch Zahlen, in Basis 8(!), als Strings gespeichert werden.

Ganz am Ende müssen noch 2 512-Byte Blöcke mit NUL-Bytes geschrieben werden, um das Archiv zu beenden.

- a) Euer Programm soll den Schlüssel `x` (extrahiere Archiv) und den Modifikator `f name` (Angabe eines Dateinamen) annehmen; diese werden beim Aufruf zu einem String kombiniert. Ist kein `f` angegeben, soll von `stdin` gelesen werden.

Aufrufe könnten also sein:

```
1 # Extrahiere Archiv "archiv.tar"
2 tar xf archiv.tar
3 # dito, lese aber ein von stdin
4 tar x
5 # *nicht* gültig
```

Da ihr bis jetzt nur Archive extrahieren könnt, sollte ein Fehler ausgegeben werden, wird der Schlüssel `x` nicht angegeben.

- b) Nun erweitert euer Programm, dass es den Schlüssel `c` (erstelle Archiv) versteht. Dieser soll ebenfalls mit `f` kombinierbar sein. Ist diesmal beim Extrahieren kein `f` angegeben, soll nach `stdout` ausgegeben werden. Die zu archivierenden Dateien folgen darauf.

Aufrufe könnten also sein:

```
1 # Archiviere folgende Dateien, und schreibe nach stdout
2 tar c datei1.txt datei2.c datei3.pdf
3 # dito, schreibe in Archiv "archiv.tar"
4 tar cf archiv.tar datei1.txt datei2.c datei3.pdf
```

- c) Erweitert euer Programm, dass es auch Ordner archivieren kann, hierzu muss es beim Erstellen in die angegebenen Ordner rekursieren (und den Ordner selber ebenfalls hinzufügen zum Archiv!). Beachtet, dass dann die `typeflag` nicht mehr `'0'` ist! Das Feld `size` sollte in aller Regel `0` sein.
- Z) Passt euer Programm so an, sodass es symbolische Links versteht, d. h. diese ebenfalls archivieren kann. Hier muss ebenfalls die `typeflag` angepasst werden. Nutzt dazu die Funktion `lstat()`. Die Dateigröße *muss* hier als `0` angegeben werden.

Tipps

- *Testet* euer Programm indem ihr mit dem vorinstallierten `tar`-Programm Archive erstellt und die von euren extrahiert oder andersherum. Da GNU/tar vorinstalliert ist, benötigt ihr möglicherweise einige Flags um Archive zu erstellen, die der obigen Beschreibung möglichst gut entsprechen:

```
1 # Die '\' lassen uns mehrzeilige Befehle schreiben
2 tar cf archiv.tar \
3   --blocking-factor 1 --format=ustar \
4   datei1.txt ...
```

- Um die von euch erstellten Archive auf Byte-Ebene mit denen von GNU/tar zu vergleichen, bietet sich es an, diese mit einem Hex-Editor o. Ä. zu betrachten, bspw. so:

```
1 # Zeige das Archiv in dem Format
2 #   pppppppp bb bb bb bb ... bb |cccccc...|
3 # an, wobei
4 # * pppppppp = Adresse,
5 # * bb = Byte in Hex,
6 # * c = Byte als ASCII
7 # und zeige das Scrollbar in dem Programm less an.
8 hexdump -v -C archiv.tar | less
```

Alternativ kann man dies mit dem etwas komplexeren Tool Radare2 machen.

- Es ist unter Linux wahrscheinlich nötig, die Präprozessor-Konstante `_POSIX_C_SOURCE` auf den Wert `200809L` zu setzen, bevor ihr Systemheader inkludiert; ggf. ist auch `_XOPEN_SOURCE` auf den Wert `500` zu setzen.
- Es ist praktisch, die Kopieroperationen mit den Funktionen `fread()` und `fwrite()` zu machen, diese aber operieren nur auf Streams, und für `fstat()` benötigt ihr Dateideskriptoren. Nutzt deshalb `fileno()` um für einen Stream den darunterliegenden Dateideskriptor zu bekommen oder mit `öffnet` die Datei mit `open()` und weißt dieser mit `fdopen()` einen Stream zu.
- Um den Nutzernamen und Gruppennamen aus der `uid` und `gid` zu bekommen, gibt es die Funktionen `getpwuid()` respektive `getgrgid()`.
- Mit `s[n]printf()` könnt ihr angenehm Oktalzahlen in die dafür vorgesehenen Felder schreiben, analog mit `sscanf()` parsen.

Weitere Links

- Der POSIX-Standard zu dem Archivierungs-Tool `pax`, welches auch `ustar` implementiert (unter „ustar Interchange Format“).
<http://pubs.opengroup.org/onlinepubs/9699919799/utilities/pax.html>
- Radare2 Hex-Editor und Reversing-Tool: <http://radare.org/>