4 Mehr zu Speicher

Wählen Sie aus den Übungsaufgaben drei Aufgaben aus.

Aufgabe 4.1. (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 4.2. (verlinkte Listen – leicht) Schreiben Sie ein Programm, dass Zahlen einließt, bis `-1` eingegeben wird. Jede Zahl soll als Glied in eine verlinkte Liste eingehängt werden. Ist die Eingabe abgeschlossen, sollen die Zahlen in umgekehrter Reihenfolge ausgegeben werden. Allozieren Sie die Glieder der Liste mit **malloc**(3). Sie können folgenden Coderahmen für die Struktur der Liste verwenden.

```c
/* Glied einer verlinkten Liste */
struct glied {
    int wert;
    struct glied *nachfolger;
};
```

Aufgabe 4.3. (Listen sortieren – mittel) Modifizieren Sie das Programm aus der vorherigen Aufgabe, sodass es die eingelesenen Zahlen sortiert ausgibt. Sortieren Sie dazu die von Ihrem Programm angelegte Liste mit dem Algorithmus *Mergesort*.

Aufgabe 4.4. (Einheitenumrechner – leicht) Schreiben Sie ein interaktives Programm, mit dem der Benutzer zwischen verschiedenen Einheiten umrechnen kann. Der Benutzer soll die Möglichkeit haben, zwei Einheiten und einen Betrag einzugeben, dass Programm gibt den Betrag dann in der Zieleinheit aus.

Denken Sie darüber nach, wie man das Programm gestalten kann, sodass eine Erweiterung auf zusätzliche Einheiten möglichst einfach ist. Hierzu ist es ggf. hilfreich, im Programm eine Tabelle anzulegen, die für jede unterstützte Einheit die Art der Einheit (z. B. Währung, Länge, Masse) und ihren Umrechnungsfaktor speichert.

Aufgabe 4.5. (Tabellen formatieren – mittel) Schreiben Sie ein Programm, das eine CSV-Datei einließt und formatiert als Tabelle ausgibt. Eine CSV-Datei ist eine Textdatei, die in jeder Zeile eine mit Kommata getrennte Liste von Feldern enthält. Die Inhalte der Felder können in Anführungszeichen gesetzt werden, wenn sie selbst Kommata enthalten, das müssen Sie aber nicht unterstützen.

```
Stadt,Einwohner,Flaeche,Land
Berlin,3613495,891.68,Berlin
Hamburg,1830584,755.22,Hamburg
Muenchen,1456039,310.70,Bayern
Koeln,1080394,405.02,Nordrhein-Westfalen
Frankfurt,746878,248.31,Hessen
Stuttgart,632743,207.35,Baden-Wuerttemberg
```

Obige Datei sollte formatiert etwa so aussehen:

```
Stadt       Einwohner  Flaeche  Land
Berlin      3613495    891.68   Berlin
Hamburg     1830584    755.22   Hamburg
Muenchen    1456039    310.70   Bayern
Koeln       1080394    405.02   Nordrhein-Westfalen
Frankfurt   746878     248.31   Hessen
Stuttgart   632743     207.35   Baden-Wuerttemberg
```

Versuchen Sie, ihr Programm so zu schreiben, dass es die Breite der Spalten an Hand des Inhaltes passend auswählt. Dazu empfiehlt es sich, den Inhalt der Datei zunächst einmal in den Arbeitsspeicher einzulesen. Im ersten Durchlauf können die nötigen Spaltenbreiten bestimmt werden, um die Spalten dann im zweiten Durchlauf optimal auszugeben.

Als Erweiterung können Sie mit den Zeichen `+`, `-`, und `|` Linien um die Zellen der Tabellen ziehen. Auch bietet es sich an, Zellen, die nur Zahlen enthalten in der Tabelle rechts statt links auszurichten.

Aufgabe 4.6. (Euler-Touren – sehr schwer) Schreiben Sie ein Programm, welches eine Euler-Tour durch einen ungerichteten Graphen findet. Lesen Sie den Graphen von der Standardeingabe als Liste von Zahlenpaaren ein. Jedes Zahlenpaar sagt Ihnen, dass es zwsichen den Knoten mit diesen Zahlen eine Kante gibt. Wie oben beenden Sie die Eingabe mit `-1`. Geben Sie die Tour als Folge von Knoten, die Sie besuchen aus. Gibt es keine Euler-Tour, so geben Sie stattdessen `-1` aus.

Recherchieren Sie mögliche Algorithmen zur Lösung des Problems und wählen Sie eine geeignete Datenstruktur. Es ist ggf. sinnvoll, den Graphen als Array von Listen adjazenter Knoten zu repräsentieren.

Aufgabe 4.7. (Worthäufigkeit – schwer) Schreiben Sie ein Programm, das Text aus der Eingabe ließt und zählt, wie oft jedes Wort in diesem vorkommt. Ist die Eingabe erschöpft, so geben Sie alle gefundenen Worte und ihre Häufigkeiten möglichst nach Häufigkeit sortiert aus. Achten Sie darauf, dass Satz- und Leerzeichen nicht Teil von Worten sind.

Machen Sie sinnvolle Annahmen über die Eingabe, z. B. dass kein Wort länger als 16 Buchstaben ist und dass es insgesamt nicht mehr als 1000 verschiedene Worte gibt. Denken Sie über eine geeignete Datenstruktur zur Ablage der Worte. Sie können zu diesem Zweck den oben angebeben Code-Rahmen um weitere Felder und Strukturen erweitern.

Aufgabe 4.8. (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 einließt, und als Datenstruktur im Computer ablegt. Sie brauchen nur PPM-Bilder vom Typ `P3` zu unterstützen.

Das Bild soll durch folgende Struktur dargestellt werden:

```c
/* ein einzelnes Pixel */
struct pixel {
    int rot, gruen, blau;
};

/* ein PPM Typ P3 Bild */
struct ppm_bild {
    /* Kopfdaten */
    int breite, hoehe;
    int max_helligkeit;

    /* mit malloc() allozieren */
    struct pixel *leinwand;
};
```

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 einließt und wieder in eine Datei schreibt. Das entstehende Bild muss identisch zum Original sein.

Aufgabe 4.9. (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, Schärfungsfilter, Kantenfilter, etc.)
- Skalierung
- Drehung um beliebige Winkel

Sie können sich auch gerne eigene Operationen ausdenken.

Aufgabe 4.10. (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 enthalt, 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 Dateiinhalt 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:

```
# Extrahiere Archiv "archiv.tar"
tar xf archiv.tar
# dito, lese aber ein von stdin
tar x
# *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:

```
# Archiviere folgende Dateien, und schreibe nach stdout
tar c datei1.txt datei2.c datei3.pdf
# dito, schreibe in Archiv "archiv.tar"
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 angebenen 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 eurem extrahiert oder andersherum. Da GNU/tar vorinstalliert ist, benötigt ihr möglicherweise einige Flags um Archive zu erstellen, die der obigen Beschreibung möglichst gut entsprechen:

```
# Die '\' lassen uns mehrzeilige Befehle schreiben
tar cf archiv.tar \
  --blocking-factor 1 --format=ustar \
  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:

```
# Zeige das Archiv in dem Format
#     pppppppp  bb bb bb bb ... bb |cccccc...|
# an, wobei
#  * pppppppp = Adresse,
#  * bb = Byte in Hex,
#  * c = Byte als ASCII
# und zeige das Scrollbar in dem Programm less an.
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 öffnet die Datei mit `open()` und weist 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/`