Programmieren 1 Übung: Zeilenweises I/O, `qsort`

Klaus Kusche

Gemeinsam für beide Übungen gilt:

- Beide Programme können mit *keinem, einem oder zwei Filenamen* aufgerufen werden. Bei keinem Filenamen sollen sie von **`stdin`** lesen und auf **`stdout`** schreiben, bei einem Filenamen von diesem File lesen und auf **`stdout`** schreiben, und bei zwei Filenamen ist der erste der Input-File und der zweite der Output-File.
- Du darfst fix eine *maximale Input-Zeilenlänge* (groß, z.B. 4096) vorgeben und sollst mit einer *Fehlermeldung* abbrechen, wenn sie überschritten wird.
- *Prüfe alle deine I/O-Operationen* (und alle *Speicher-Allokationen*) auf Fehler und gib ordentliche *Fehlermeldungen* aus!

1.) Wortersetzung

Dein Programm wird mit *zwei Worten* (gefolgt von keinem, einem oder zwei *Filenamen*) auf der Befehlszeile aufgerufen. Es soll den Input auf den Output kopieren und dabei *alle Vorkommen des ersten Wortes durch das zweite Wort ersetzen*, egal, ob das Wort in einer Zeile gar nicht, einmal oder mehrmals vorkommt.

Lösungsidee:

- Der Input wird *zeilenweise* verarbeitet.
- In jeder Zeile suchen wir das *erste Vorkommen* des gesuchten Wortes (dafür gibt es eine String-Funktion, verwenden!).
- An der Stelle, an der das gesuchte Wort anfängt, *beenden* wir den bisherigen Text der Zeile (siehe unten!), und *geben den Teil bis dorthin aus* (ohne Zeilenvorschub!).
- Gleich anschließend *geben wir das Ersatzwort aus*.
- Dann *suchen wir wieder* nach einem Vorkommen des gesuchten Wortes, und zwar ab dem Zeichen der Zeile, das auf das gerade bearbeitete Vorkommen des Wortes folgt (d.h. wir überspringen das Wort).
- Das wiederholen wir, bis das Wort im Rest der Zeile *nicht mehr vorkommt*. Dann wird der *gesamte Rest der Zeile* (ab der Stelle, wo die letzte Suche begonnen hat) *ausgegeben*.

Tipp:

- Nimm das in der Stunde besprochene File-I/O-Beispiel als Ausgangsbasis!
- Zum Verständnis der Logik kannst du auch einen Blick auf die Musterlösung des String-Replace-Beispiels (mit Stringfunktionen, nicht zeichenweise) werfen.
- Überlege: Wenn du aus dem vorderen Teil eines Strings einen eigenen String machen möchtest und das Zeichen, vor dem der neue String enden soll, nicht mehr brauchst: Wie kannst du den String am einfachsten "teilen"?
- Du wirst mit **`char`**-Pointern arbeiten müssen.

2.) Textfile sortieren

Dein Programm wird mit keinem, einem oder zwei Filenamen (siehe oben) aufgerufen. Der Output soll dieselben Zeilen enthalten wie der Input, aber *zeilenweise alphabetisch sortiert* (gemäß der Sortier-Reihenfolge, die **`strcmp`** intern verwendet).

Lösungsidee:

- Zu diesem Zweck müssen wir zuerst einmal den *gesamten File in den Speicher laden*. Das soll nicht mehr Speicher brauchen, als die Zeilen wirklich lang sind. Verwende daher *keine zweidimensionale Matrix* mit fixer Länge für die Zeilen, sondern lies eine Zeile nach der anderen in denselben fixen String und mach dann *von jeder gelesenen Zeile eine dynamisch angelegte Kopie* (dafür gibt es eine *vordefinierte String-Funktion!*).
- Weiters brauchst du ein *Array von Pointern, das auf diese dynamisch angelegten Zeilen zeigt* und beim Lesen der Reihe nach befüllt wird.

  Auch hier soll es *keine fixe Größenbeschränkung* geben:
  Wir beginnen mit einem *dynamisch angelegten Array* für 100 Zeilen,
  und jedesmal, wenn der Platz erschöpft ist, *verdoppeln wir die Größe* des Arrays (welche Allokations-Funktion ist dazu sinnvoll?).

  Du wirst dir dazu sowohl die momentan *allokierte Größe* als auch den *aktuellen Füllstand* (Anzahl der belegten Zeilen) des Arrays merken müssen.
- Wenn wir alle Zeilen gelesen haben, *sortieren wir dieses Array* von Pointern auf die Zeilentexte mit der *eingebauten Funktion* **`qsort`** (es werden *nur die Pointer* im Array umgeordnet, die Zeilentexte selbst bleiben unberührt!).

  Wie man **`qsort`** anwendet (und wie die *Vergleichsfunktion*, die man dafür braucht, ungefähr aussieht), kannst du z.B. dem Beispiel auf der Linux-Man-Page von **`qsort`** entnehmen:

  - **`qsort`** braucht einen Pointer auf das zu sortierende Array, die Anzahl der Elemente darin, die Größe eines Elementes in Bytes, und einen *Pointer auf die Funktion*, die zum Vergleich zweier Elemente aufgerufen werden soll.
  - Die Vergleichsfunktion wird mit *zwei Pointern auf die zu vergleichenden Array-Elemente* (als **`void *`**) aufgerufen und soll so wie **`strcmp`** eine Zahl kleiner 0, 0 oder größer 0 liefern.

    In unserem Fall sind das also zwei *Pointer auf* **`char`**-*Pointer* (casten!). Diese Pointer zeigen auf zwei Array-Elemente, und wir müssen die beiden Strings, auf die diese beiden Array-Elemente zeigen, mit **`strcmp`** vergleichen.
- Als letzten Schritt geht man das sortierte Array einmal durch und *gibt alle Zeilen der Reihe nach aus*.

Schrittweises Vorgehen:

Wenn du dir nicht alles auf einmal zutraust, geh in folgenden Schritten vor:

- Verzichte zuerst einmal auf das Sortieren und das dynamisch wachsende Array: Leg ein lokales Array fixer Größe von **`char`**-Pointern an.

Lies die Datei wie oben beschrieben zeilenweise und speichere die Pointer
auf die dynamischen Kopien der gelesenen Zeilen in diesem Array
(brich mit einer Fehlermeldung ab, wenn das Array zu klein ist).
Geh dann das Array mit einer zweiten Schleife nochmals durch
und gib die Datei unverändert wieder aus.

- Als nächstes könntest du versuchen, das Array zwischen den beiden Schleifen mit **`qsort`** zu sortieren (die nächsten Schritte sind aber unabhängig vom **`qsort`**).
- Ersetze dann das lokale Array durch ein mit **`malloc`** dynamisch angelegtes Array fixer Größe.
- Erst wenn das klappt: Versuche das Array bei Bedarf mit **`realloc`** zu vergrößern.