Programmieren C: `qsort`-Funktion, Strukturen, `malloc`, File-I/O: Text sortieren

Klaus Kusche

Gesucht ist ein Programm, das im Endausbau Textdateien *zeilenweise sortiert* (*alphabetisch* in der Ordnung, die **`strcmp`** liefert). Wir gehen dabei schrittweise vor.

1.) Sortieren: `qsort`

Echtes File-I/O wollen wir erst ganz am Ende dazubauen.
In der ersten Ausbaustufe liest das Programm einfach *zeilenweise vom Terminal* (mit **`fgets`**) und schreibt auch zeilenweise auf das Terminal (mit **`fputs`**).

Für diejenige, die kein File-I/O kennen:

- **`fgets(`*`line`*`, `*`maxlen`*`, stdin)`** liest *eine Zeile* in das **`char`**-Array ***line*** ein
  (bei uns wird ***line*** anfangs die **`i`**-te Zeile eines zweidimensionalen Arrays sein),
  wobei das Array ***maxlen*** Zeichen groß ist (***maxlen*** ist ein **`int`**,
  und es werden maximal ***maxlen-1*** Zeichen gelesen).
  `stdin` steht für das Terminal (**`fgets`** könnte auch aus einer Datei lesen).

  Das **`fgets`** speichert auch das **`\n`** am Ende der Zeile im Array ***line***,
  du brauchst also beim Ausgeben der sortierten Zeilen kein **`\n`** mehr anhängen.

  Wenn die Zeile zu lang war, enthält ***line*** *kein* **`\n`** . Prüfe das nach jedem **`fgets`**;
  gib eine Fehlermeldung aus und beende das Programm, wenn das **`\n`** fehlt.

  Achtung: Editiere deine Testdaten-Datei so, dass auch die *letzte Zeile* mit **`\n`** endet,
  denn unter Windows werden Texte meist *ohne* **`\n`** in der letzten Zeile gespeichert!

  Als *Returnwert* liefert **`fgets`** einen Pointer auf ***line*** oder den **`NULL`**-Pointer,
  wenn der Input zu Ende ist. In diesem Fall muss deine Lese-Schleife *enden*.
- **`fputs(`*`line`*`, stdout)`** gibt den Text ***line*** *am Terminal aus*.

Zum Testen:

- Das Programm kannst du am besten testen, indem du es im DOS-Fenster startest
  und dabei mit **<** eine *Textdatei in die Eingabe des Programms umleitest*:

  `textsort < eingabe.txt`

  Analog kannst du mit **>** auch die *Bildschirm-Ausgabe in eine Datei umleiten*:

  `textsort < eingabe.txt > ausgabe.txt`
- Wenn du den *Input direkt eintippst*, kannst du den *Input beenden*
  (d.h. ***„End of File“*** eingeben), indem du *am Anfang einer neuen Zeile*
  `Strg+Z` gefolgt von **`Enter`** tippst (unter *Linux* und am *Mac*: **`Strg+D`**).

Im ersten Schritt speichern wir den gesamten File-Inhalt zeilenweise in einem
fix dimensionierten zweidimensionalen **`char`**-*Array* (vorderer Index = Zeilennummer).

Die *maximale Zeilenanzahl* des Inputs (= Zeilenanzahl des Arrays) und
die *maximale Zeilenlänge* einer Input-Zeile (= Spaltenanzahl des Arrays)
darfst du *fix vorgeben*, mach *schöne benannte Konstanten* dafür!
(beachte, das ein *lokales Array nicht allzu groß* sein darf)

Das *Hauptprogramm* besteht aus *drei* Schritten:

- Dem *zeilenweisen Einlesen* des Textes in das Array:
  Lies in einer *Schleife* (*mitzählen!*) mit **`fgets`** jeweils die *nächste Zeile* der Eingabe in die nächste Zeile des zweidiemnsionalen Arrays
  (ein zweidimensionales Array mit nur *einer* **`[ ]`** dahinter liefert einen *Pointer* auf die entsprechende *Zeile des Arrays*, diese Zeile kann *wie ein normales Array*, d.h. in unserem Fall wie ein String, verwendet werden).

  Die Schleife wird normalerweise *verlassen*, wenn **`fgets`** das *Dateiende* meldet.

  Wenn unser Array *voll* ist (*maximale Zeilenzahl* erreicht) oder die *Zeile zu lang* war, muss dein Programm ebenfalls *enden* (gib eine *Fehlermeldung* aus!).
- Dem *Sortieren* des Arrays (**`qsort`**-Aufruf, *siehe unten!*).
- Dem *zeilenweisen Ausgeben* des sortierten Arrays:
  Schreib eine *Schleife* über alle *belegten* Elemente des Arrays
  und gib jede Zeile mit **`fputs`** aus.

Versuche, zuerst das Hauptprogramm *nur mit Einlesen und Ausgeben* (*ohne* Sortieren) zum Laufen zu bringen. Bau das Sortieren erst dazu, *nachdem* das klappt.

Für das *Sortieren* verwenden wir diesmal die *vordefinierte Funktion* **`qsort`** aus **`stdlib.h`**.
Diese Funktion wird mit *4 Argumenten* aufgerufen:

- Dem zu sortierenden *Array* (unserem zweidimensionalen Array).
- Der *Anzahl* der (belegten!) Elemente im Array,
  d.h. der Anzahl der tatsächlich *eingelesenen* Zeilen.
- Der *Größe eines Elementes* (also der Größe *einer Zeile* unseres Arrays) in *Bytes*
  (maximale Zeilenlänge, *ein* **`char`** ist *ein* Byte lang).
- Der zu verwendenden *Vergleichsfunktion* (*nur der Name* der Funktion,
  ohne **`( )`** dahinter, denn die Funktion wird an dieser Stelle ja nicht aufgerufen,
  sondern dem **`qsort`** wird ein *Pointer auf die Funktion übergeben*).

`qsort` kennt das „Innenleben“ der Array-Elemente nicht:
Für **`qsort`** ist jedes Element einfach eine fixe Anzahl von Bytes.
`qsort` weiß daher auch nicht, wie man zwei Elemente vergleicht,
sondern ruft dafür die Vergleichsfunktion auf, die man ihm als 4. Argument übergibt.

Diese *Vergleichsfunktion müssen wir selbst schreiben*.
Sie muss wie folgt aussehen und funktionieren:

- *Als Parameter hinein* gehen zwei *Pointer auf die zu vergleichenden Elemente*.
  Diese müssen beide als „**`const void *`**“ deklariert werden,
  also als Pointer auf Daten unbekannten Typs, die nicht verändert werden dürfen.

  Bei uns zeigen die Pointer jeweils auf eine Zeile des Arrays, also auf einen String.
- *Als Returnwert* muss die Vergleichsfunktion dasselbe wie **`strcmp`** liefern:
  Einen **`int`**, und zwar eine Zahl **`<0`**, wenn das *erste* Argument das kleinere ist,
  eine Zahl **`>0`**, wenn das *zweite* Argument das kleinere ist,
  und **`0`**, wenn beide Elemente betreffend Sortierung *gleichgroß* sind.

Also müssen wir die beiden Parameter intern in zwei **`const char`**-Pointer *umwandeln*, und die Texte, auf die diese Pointer zeigen, mit **`strcmp`** vergleichen.
Das Ergebnis von **`strcmp`** wird als Returnwert unserer Funktion zurückgegeben.

2.) Zeilennummern: Strukturen

Im nächsten Schritt soll das Programm bei der Ausgabe vor jede Zeile
die ursprüngliche Zeilennummer der Zeile im Input (ab 1 gezählt, nicht ab 0) schreiben.
Man muss diese Nummer beim Einlesen mit der Zeile speichern und mitsortieren.

Im Detail sind folgende Schritte notwendig:

- Wir brauchen einen *Strukturtyp* für die Speicherung einer *einzelnen Zeile*.
  Er muss einen **`int`** für die *Zeilennummer* und ein **`char`**-Array für den *Zeilentext* enthalten (Arraygröße = maximale Zeilenlänge).

  Deklariere den Typ so, dass du ihn im Rest des Programmes
  ohne **`struct`** verwenden kannst!
- Das Hauptprogramm bekommt statt dem zweidimensionalen Array jetzt
  ein Array solcher Strukturen (Arraygröße = maximale Zeilen-Anzahl).
- Beim *Einlesen* muss das **`fgets`** die Zeile jetzt *in das Text-Member der Struktur* speichern (und bekommt natürlich auch *dessen Größe* als Längen-Limit).
  Weiters muss die *aktuelle Zeilennummer* im *Nummern-Member* gespeichert werden.
- In der *Ausgabe* wird aus dem **`fputs`** ein **`printf`** mit *Zeilennummer und Zeilentext*
  (kannst Du die Zeilennummer *fix mit 8 Zeichen Breite* und *führenden 0* ausgeben?).
- Beim **`qsort`** muss als *Elementgröße* die *Größe einer Struktur* angegeben werden.
- Die *zwei Array-Element-Pointer,* die unsere *Vergleichsfunktion* bekommt,
  sind jetzt in Wirklichkeit *Pointer auf (konstante) Strukturen*.
  Das **`strcmp`** wird *für die Text-Member* dieser beiden Strukturen aufgerufen.

3.) Dynamische Zeilenlänge: `strdup`

In unserem Programm wird bisher *für jede Zeile Platz in derselben, fixen Länge* reserviert.
Diese Länge beschränkt zugleich die Länge der Input-Zeilen und ist schwierig festzulegen:
Macht man sie zu groß, wird bei langen Dateien *sehr viel Speicherplatz verschwendet*.
Macht man sie zu klein, lassen sich *Dateien mit längeren Zeilen nicht mehr sortieren*.

Wir wollen daher für jede Zeile in unserem Array dynamisch *nur so viel Platz* anlegen,
wie die *Zeile wirklich lang* ist.

Damit wir die anzulegende Länge feststellen können,
müssen wir jede Zeile allerdings zuerst einmal *in ein Hilfsarray fixer Länge einlesen*.
Es gibt also weiterhin eine *fixe Obergrenze* für die Zeilenlänge.
Diese kann allerdings *viel größer* gewählt werden (z.B. 4 KB oder 64 KB), weil ja nur
ein einziger String dieser Größe fix angelegt wird und *nicht ein ganzes Array* von Strings.

Im C-Standard gibt es *keine* Funktion, die eine *beliebig lange Zeile* einlesen kann.
Man müsste die Zeile *zeichenweise einlesen,* und zwar in ein *dynamisch angelegtes Array,*
das jedesmal *dynamisch vergrößert* wird, wenn es *zu kurz* wird
(analog zur Vorgehensweise im *nächsten Punkt*),
und das nach dem Einlesen wieder *auf die tatsächliche Länge der Zeile gekürzt* wird.

Den Aufwand für dieses letzte Stückchen Flexibilität wollen wir uns aber sparen,
zumal eine derartige Funktion für lange Zeilen relativ Laufzeit-ineffizient ist
(es gäbe auch fertige, aber nicht standardisierte Funktionen, die genau das tun).

Dafür sind folgende Schritte nötig:

- Wir erhöhen die *maximale Zeilenlänge*,
  legen im Hauptprogramm einen *Hilfs-String* dieser Länge an,
  und lesen beim **`fgets`** *in diesen String* ein.
- Falls es bei dir nicht ohnehin schon dort ist,
  kann das **`fgets`** jetzt *direkt in die Schleifenbedingung* wandern:
  Ab jetzt können wir ja immer versuchen, in den Hilfs-String einzulesen,
  ohne schon vorher zu prüfen, ob noch Platz im Array ist.
  Erst *nachdem* wir erfolgreich gelesen haben, prüfen wir,
  ob im Array noch ein Element für die soeben eingelesene Zeile frei ist.
- In unserem *Struktur-Typ* ersetzen wir das **`char`**-*Array* durch einen **`char`**-*Pointer*.
- *Nach dem Lesen jeder Zeile* erstellen wir mit **`strdup`** eine *dynamisch angelegte Kopie*
  des eingelesenen Strings und *speichern einen Pointer auf diese Kopie*
  im Pointer-Member der aktuellen Struktur.

Durch diese Änderung wird unser Programm auch *viel schneller*,
weil in den Array-Elementen *nur mehr Pointer* auf Texte stehen und *nicht die Texte selbst*.
Dadurch sind die Elemente *viel kleiner* und können beim Sortieren *viel schneller vertauscht* werden. Bisher wurden beim Sortieren auch die Texte selbst herumkopiert,
jetzt bleiben die *Zeilentexte beim Sortieren unverändert* an derselben Stelle im Speicher.

Da das **`strdup`** (und später das **`malloc`** usw.) ja *scheitern* könnte,
schreiben wir noch eine kleine *Funktion zur Fehlerprüfung*:

- Sie bekommt einen *„Pointer auf irgendetwas“* als Parameter.
  Ist dieser Pointer **`NULL`**, *beendet* sie das Programm mit einer *Fehlermeldung*.
  Ansonsten gibt sie den Pointer *unverändert als Returnwert zurück*
  (*welchen Returntyp* hat diese Funktion daher?).
- Für die Ausgabe einer schönen Fehlermeldung in dieser Funktion ist es sinnvoll,
  eine *globale Variable für den Programmnamen* anzulegen
  und zu Beginn des Hauptprogrammes zu befüllen.
- Im *Hauptprogramm* werden dann alle **`strdup`**- und später **`malloc`**-Aufrufe
  in einen Aufruf dieser Funktion *„verpackt“*:
  Der **`strdup`**-Aufruf steht *als Argument* im Aufruf der Fehlerprüf-Funktion,
  und der fehlergeprüfte Returnwert der Funktion
  wird im Text-Pointer unserer Struktur gespeichert.

Und weil wir gerade so schön mit Pointern arbeiten, noch eine *kleine Pointer-Übung*:
Bau die *Schleifenvariable der Ausgabe-Schleife* von einem **`int`** auf einen **`struct`**-*Pointer* um, der durch das Array läuft. Die Werte im **`printf`** sollten dadurch *einfacher* werden.

4.) Dynamische Zeilenanzahl: `malloc`, `realloc`

Jetzt wollen wir das *zweite fix codierte Limit* aus unserem Programm *entfernen*:
Die harte Grenze für die *Anzahl der Zeilen* im zu sortierenden Input.

Die Idee ist folgende:

- Wir deklarieren unser *Array von Strukturen* nicht mehr fix,
  sondern legen es *dynamisch* an, und zwar *vorläufig eher klein* (z.B. 100 Strukturen).
- *Jedesmal,* wenn im Array beim Lesen einer Zeile *kein Platz mehr ist,*
  verdoppeln wir die Größe des bestehenden Arrays
  (der Grund für die Verdopplung ist, dass eine Größenänderung *eher aufwändig* ist
  und daher nicht zu oft erfolgen sollte, jedenfalls nicht bei jeder einzelnen Zeile.
  Man macht das Array daher auf Verdacht immer gleich reichlich größer).

Es sind folgende Änderungen nötig:

- Aus der *Konstante* für die maximale Zeilen-Anzahl wird
  eine Konstante für die *beim ersten* **`malloc`** *angelegte Zeilen-Anzahl*.
- Aus dem **`struct`**-Array in unserem Hauptprogramm wird
  ein *Pointer* auf das dynamisch angelegte **`struct`**-Array.
- Zusätzlich zur Anzahl der gerade belegten Array-Elemente
  brauchen wir einen *zweiten* **`int`** für die *aktuelle Größe des Arrays*
  (= Anzahl der insgesamt *angelegten Elemente*).
- *Vor der Lese-Schleife* wird mit **`malloc`** ein Array der vorgegebenen Größe *angelegt*.
  (wie berechnest du aus der Konstante die Größe für das **`malloc`**?)
  Vergiss nicht, dir die aktuelle Größe zu merken und den **`malloc`**-Aufruf zu prüfen!
- Die bisherige *„Array-Voll-Prüfung“* wird umgebaut:
  Vor dem Speichern der soeben gelesenen Zeile im Array prüfen wir,
  ob im Array *noch ein Element frei* ist.
  Wenn nicht, *vergrößern* wir unser Array:
  Wir *verdoppeln* unsere Variable für die aktuelle Größe
  und passen das bestehende Array mit **`realloc`** an die neue Größe an.

  Achtung: Das **`realloc`** wird das Array in vielen Fällen *verschieben,*
  d.h. es liefert einen *neuen Pointer* auf den Array-Anfang!

  Prüfe den **`realloc`**-Aufruf wieder mit unserer Hilfsfunktion auf „kein Platz“!

5.) File-I/O

Als letzten Schritt bauen wir unser Programm wie gewohnt so um,
dass es mit Filenamen auf der Befehlszeile aufgerufen werden kann:

- Das Programm kann mit *keinem, einem oder zwei Filenamen* aufgerufen werden.
  Bei *keinem* Filenamen soll es 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.
- *Prüfe alle deine I/O-Operationen* auf Fehler (nicht nur Öffnen und Schließen,
  sondern auch Lesen und Schreiben) und gib ordentliche *Fehlermeldungen* aus
  (in der Art meiner Hilfsfunktion, die ich in der Vorlesung gezeigt habe)!

6.) Beliebige Zeilenlänge

Die einzige große Schwäche, die unser Programm für einen praktischen Einsatz noch hat, ist die *Begrenzung der Länge der Input-Zeilen* mittels Compilezeit-Konstante.

Da es im C-Standard *keine* Funktion gibt, die beliebig lange Zeilen einliest[1], müssen wir das selbst programmieren. Wir schreiben dafür eine *eigene Funktion* **`readLine`**.
Sie bekommt den *Input-File als Parameter*, liest daraus *eine beliebig lange Zeile* incl. **`\n`** und returniert einen *Pointer auf einen frisch dynamisch angelegten String* mit dieser Zeile.

Diese Funktion enthält wie bisher einen *String fixer Länge* und befüllt diesen mit **`fgets`**.
Wenn schon dieses erste **`fgets`** *fehlschlägt*, returniert die Funktion sofort **`NULL`**.
Ansonsten erzeugt sie mittels **`strdup`** (auf Fehler *prüfen*!) eine *dynamische Kopie* des gelesenen Strings als Ergebnis.

Dann folgt eine *Schleife*: Solange der String noch *kein* **`\n`** enthält, wird *immer wieder*
- mit **`fgets`** *weiterer Text gelesen* (jedesmal in *denselben String* fixer Länge),
- die *neue Länge* des Ergebnisses *berechnet* (*Summe* der Längen des bisherigen Ergebnisses und des frisch gelesenen Strings, was darfst du dabei nicht vergessen?),
- der bisherige String mit **`realloc`** *dynamisch vergrößert* (da **`strdup`** intern ein **`malloc`** macht, darf man ein **`strdup`**-Ergebnis mit **`realloc`** vergrößern),
- und der *neu gelesene Teil* hinten an den bisherigen String *angehängt*.

Es gibt noch einen *Sonderfall*: Sollte das wiederholte **`fgets`** *scheitern*, muss die Funktion statt dem neu gelesenen String ein **`"\n"`** an das Ergebnis *anhängen*, weil die Zeile *immer* mit **`\n`** enden soll (auch dazu muss man das *Ergebnis vergrößern*).

Je nach Geschmack kann man diese Lösung *in zwei verschiedene Richtungen verbessern*:

- *Schönheitsfreaks* werden bemängeln, dass das *Anhängen* eines Textes an einen bestehenden dynamischen String (zwei Mal **`strlen`**, **`realloc`** und **`strcat`**) in eine *separate, universell verwendbare Funktion* ähnlich **`strdup`** ausgelagert werden sollte.
- *Geschwindigkeitsfreaks* werden bemängeln, dass der bisherige Ergebnis-String *in jedem Schleifendurchlauf ohne Notwendigkeit drei Mal komplett durchlaufen* wird:
  - Einmal vom **`strchr`** in der **`while`**-Bedingung bei der *Suche* nach einem **`\n`**.
  - Einmal vom **`strlen`** beim Berechnen der neuen *Gesamtlänge*.
  - Und einmal vom **`strcat`** bei der Suche nach dem *Ende*, an das angehängt wird.

  Wenn man mit *ein oder zwei zusätzlichen Variablen* über die *aktuelle und neue Länge* des Ergebnis-Strings Buch führt, dann lassen sich *alle drei Durchläufe* des bisherigen Strings *vermeiden*.

1 Im *POSIX-Standard* gibt es seit 2008 eine Funktion **`getline`**, die genau das leistet, aber bisher gibt es keine Absichten, diese Funktion in den C-Standard zu übernehmen.