3 C-Syntax II

Sie können auch die Aufgaben der letzten Aufgabenzettel bearbeiten.

Aufgabe 3.1. (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 3.2. (Cache-Lokalität – leicht) Ein CPU-Cache kann konsekutive Bytes in n-Byte–Blöcken aus dem Hauptspeicher vorlagern, um den Zugriff zu beschleunigen.

Schreibt ein Programm welches

- eine $n \times m$-Matrix initalisiert
- die Werte darin aufsummiert, erst die Spalten, dann die Zeilen
- die Werte darin aufsummiert, erst die Zeilen, dann die Spalten

Welche Version ist schneller? Warum (wie sieht die Matrix im Speicher aus)?

Die Arrays sollten sehr groß sein, um einen Effekt zu messen

Aufgabe 3.3. (Überlauf von time_t – leicht) In der Manpage `tunefs(8)` gibt/gab es das Kommentar „Take this out and a Unix Daemon will dog your steps from now until the `time_t`'s wrap around“ zu dem Satz „You can tune a file system, but you cannot tune a fish“ (ein Daemon ist ein Hintergrundprogramm). Das `time_t` bezieht sich auf einen Datentyp aus `time.h`, welcher die aktuelle POSIX oder UNIX Epoch time hält, d. i. die Sekunden seit dem 1.1.1970.

Die Aufgabe ist: Schreibt ein Programm, welches berechnet, wann dieser Datentyp nicht mehr die aktuelle Zeit darstellen kann.

Aufgabe 3.4. (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. Sie können folgenden Coderahmen für die Struktur der Liste verwenden.

```c
#include <stdlib.h>

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

/* Vorrat von Listenelementen */
```

```
10 #define VORRATSGROESSE 1000
11 struct glied vorrat[VORRATSGROESSE];
12 int verbrauch = 0;
13
14 /* Gib mir ein neues Glied aus dem Vorrat */
15 struct glied *alloziere_glied(void)
16 {
17     if (verbrauch >= VORRATSGROESSE) {
18         fprintf(stderr, "Gliedervorrat␣leer!\n");
19         exit(1);
20     }
21
22     return (&vorrat[verbrauch++]);
23 }
```

Aufgabe 3.5. (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 3.6. (Stringsuche – mittel) Schreiben Sie Ihre eigene Variante der Funktion **strstr**(3). Schreiben Sie ein Programm, um diese zu testen. Hierzu bietet es sich an, dass Ihre eigene Implementierung mit der von System bereitgestellten Implementierung zu vergleich.

Analysieren Sie die Laufzeit Ihrer Implementierung und recherchieren Sie bessere Algorithmen zur Suche von Strings in Strings.

Aufgabe 3.7. (env(1) – mittel) Schreiben Sie eine Implementierung des Programmes **env**(1). Dieses hat zwei Aufgaben:

Ohne Argumente aufgerufen, gibt **env**(1) den Inhalt der zur Zeit gesetzten Umgebungsvariablen aus. Diese können Sie durch Inspektion der globalen Variable **environ**(7) auslesen.

Werden Argumente übergeben, so wird jedes Argument der Form

Schluessel=Wert

mit **putenv**(3) als Variable in die Umgebung aufgenommen. Alle Argumente ab dem ersten Argument, das nicht diese Form hat, werden als Name und Argumente eines auszuführenden Programmes verstanden, welches nach Setzen der Umgebungsvariablen mit **execvp**(2) ausgeführt wird.

Weitere Funktionalität von **env**(1) ist nicht Teil der Aufgabe.

Aufgabe 3.8. (Stack als Array – mittel) Ein Stack ist eine Datenstruktur auf der man auf das oberste Element zugreifen kann: Man kann es löschen (POP), angucken (PEEK) oder ein anderes Element „darüber legen“ (PUSH).

Implementiert diese Funktionalität mithilfe einer Struktur in welcher die Daten in einem Array gespeichert werden. Als Idee: Die Struktur benötigt mindestens Einträge um 1. die aktuelle Position des Stacks zu speichern und 2. zu wissen, ob der Stack leer oder voll ist. Diese wird dann als Pointer an die Funktionen übergeben, samt möglicherweise weiteren benötigten Argumenten.

Aufgabe 3.9. (Nutzung des Stacks: Taschenrechner – schwer) Entwickeln Sie einen Taschenrechner der mit sogenannter umgekehrter polnischer Notation bzw. Postfix-Notation arbeitet, d. h. an Stelle von $a + b$ würde man schreiben $a\ b\ +$. Das Praktische dabei ist, dass man Klammern komplett weglassen kann. Dies wird deutlich, schreibt man unterschiedlich-geklammerte Ausdrücke in PN:

$$(2 + 3) \cdot 4 \qquad 2\ 3\ +\ 4\ \cdot$$
$$2 + (3 \cdot 4) \qquad 4\ 3\ \cdot\ 2\ +$$

Die Idee bei der Implementierung ist einen Stack zu nutzen:

- Wird eine Zahl gelesen, wird diese auf den Stack gepusht
- Wird ein Operator gelesen, werden n Elemente vom Stack gepopt, wobei n die Anzahl der Operanden ist, die dieser Operator benötigt. Das Ergebnis wird berechnet und wieder auf den Stack gepusht.

Am Ende ist das Ergebnis das einzige Elemente auf dem Stack.

Testet den Algorithmus erst von Hand um euch die Funktionsweise klar zu machen!

Implementation: Ihr könnt die Eingabe entweder über Programmargumente annehmen (etwas einfacher), oder direkt von `stdin` einlesen. In letzterem Fall habt ihr die Möglichkeit euer Programm auch so zu erweitern, dass nach einer erfolgreichen Berechnung ihr weiterrechnen könnt und nicht das Programm erneut starten müsst. Das ist gerade dann sinnvoll, wenn ihr später Variablen einbaut, das heißt ihr könnt Ergebnisse zwischenspeichern um sie später wiederzubenutzen.

Aufgabe 3.10. (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 3.11. (base64 – schwer) Schreiben Sie zwei Programme, die Dateien im Base64-Format kodieren bzw. aus dem Base64-Format dekodieren. Recherchieren Sie dazu wie das Format Base64 funktioniert. Sie können die Korrektheit Ihrer Implementierung gegen das Linux-Programm `base64` testen.

Aufgabe 3.12. (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.