Institut für Informatik
Prof. Dr. Stefan Conrad

Sommersemester 2021
Übung 6

Hardwarenahe Programmierung
Gruppe 17 (Florian)

Es ist soweit: In dieser Übung nutzen Sie dynamische Speicherverwaltung in C! Außerderm verwenden Sie `valgrind`*, um Ihre Programme auf Speicherlecks zu untersuchen.*

- *Denken Sie wie immer daran, Ihre Lösungen im ILIAS hochzuladen und den Test im ILIAS zu absolvieren!*
- *Für das Abschlussprojekt müssen Sie dynamische Speicherverwaltung beherrschen!*

Aufgabe 1 *Tiere (**Pflichtaufgabe**)*

In dieser Aufgabe sollen Tiere mit Hilfe einer verketteten Liste gezählt werden. Die Tiernamen enthalten keine Leerzeichen und sind weniger als 15 Zeichen lang:

```
Fluffy
Pluto
Snoopy
Fluffy
```

(a) Die Tiernamen sollen aus einer gegebenen Eingabedatei eingelesen und nacheinander an das Ende einer verketteten Liste angehangen werden. Für die einzelnen Listenknoten soll das Struct `Tier` verwendet werden, wofür Sie sich sinnvolle Attribute überlegen müssen. Tiernamen, die doppelt vorkommen, sollen nicht nochmals angehangen werden. Stattdessen soll ein Zähler im entsprechenden Listenknoten zählen, wie häufig dieser Name vorkam.

(b) Danach sollen die Häufigkeiten und Namen in eine gegebene Ausgabedatei ausgegeben werden:

```
2 Fluffy
1 Pluto
1 Snoopy
```

(c) Implementieren Sie dazu die nötigen Funktionen in `liste.c`. Die `main`-Funktion ist Ihnen in der Datei `liste.c` schon vorgegeben und darf nicht verändert werden!

(d) Der Speicher der Listenknoten muss dabei dynamisch mit `malloc` und `free` verwaltet werden. Das bedeutet, dass Ihr Programm eine Liste mit beliebig vielen Tieren verarbeiten können muss.

(e) Denken Sie daran, alle Variablen in Ihren Listenknoten vor der Verwendung zu initialisieren.

(f) Abschließend muss der allozierte Speicher freigegeben werden.

(g) Denken Sie daran, Variablen nach der Freigabe nicht mehr zu verwenden.

(h) Nutzen Sie `valgrind`, um ihre Speicherverwaltung zu überprüfen.

(i) Testen Sie Ihr Programm mit dem Test-Skript `test.sh`.

Die Aufgabe **muss** mit einer verketteten Liste gelöst werden. Andere Lösungen, die zum Beispiel ein Array von Structs anstelle einer verketteten Liste verwenden, werden nicht akzeptiert.

Aufgabe 2 *Text (**Pflichtaufgabe**)*

In dieser Aufgabe sollen Sie eine Datenstruktur erstellen, die einen Text verwaltet. Der Speicher für den Text muss dazu dynamisch alloziert werden, d.h. es darf keine feste Obergrenze für die Größe geben.

(a) Implementieren Sie in der Datei `text.h` ein `struct` mit sinnvollen Einträgen.

(b) Implementieren Sie in der Datei `text.c` eine Funktion, die ein `struct` vom Typ `Text` erzeugt und mit einem String initialisiert sowie eine weitere Funktion, die den allozierten Speicher wieder freigibt.

```
Text* text_initialisieren(char *string);
void text_freigeben(Text *text);
```

Sehen Sie sich die Header-Datei und die Tests an, um zu verstehen, wie diese Funktionen verwendet werden.

(c) Implementieren Sie eine Funktion, die Speicher für einen Ausgabe-String alloziert und den im `struct` gespeicherten Text in diesen String schreibt.

```
char* text_ausgeben(Text *text);
```

Für das Freigeben des Ausgabe-Strings ist die aufrufende Funktion verantwortlich.

(d) Implementieren Sie eine Funktion, die einen Text anhand eines Strings (mit Länge größer 0) in weitere Text-Structs trennt und Zeiger auf Zeiger auf die so entstandenen Text-Stücke zurückgibt. Die Anzahl dieser Stücke soll über den Zeiger `anzahl` zurückgegeben werden.

```
Text** text_trennen(int *anzahl, Text *text, char *trenner);
```

Zum Beispiel würde der Text „`12ggg34gg`“ mit Trennstring „`gg`“ zu den drei Texten „`12`“, „`g34`“ und „“ zertrennt werden. Beachten Sie, dass Texte der Länge 0 bei Trennstrings am Anfang oder Ende des Eingabetextes entstehen.

Die zugehörige Funktion `texte_freigeben`, die die Texte wieder freigibt, ist bereits für Sie implementiert.

(e) Überprüfen Sie Ihr Programm mit `valgrind` und `make run`. Die Speicherlecks der Kategorie `still reachable`, die durch das `check`-Framework verursacht werden, dürfen hierbei ausnahmsweise ignoriert werden.

Aufgabe 3 *Statische Code-Analyse (Vorbereitungsaufgabe)*

Das Programmieren in C kann sehr fehleranfällig sein. Aus diesem Grund gibt es eine Vielzahl an Programmen, die Code auf häufige Fehler überprüfen.

Sie erhalten die Datei `main.c`. Das hieraus resultierende Programm liest zwei Ganzzahlen von der Standardeingabe ein, generiert alle Ganzzahlen dazwischen und gibt diese aus. Es wird nicht überprüft, ob sinnvolle Eingaben getätigt wurden und zudem gibt es einige Fehler.

Installieren Sie eins (oder sogar alle) der Programme zur statischen Code-Analyse und führen Sie das Programm aus, um Fehler zu finden.

Hierbei handelt es sich nicht um eine vollständige Liste. Recherchieren Sie nach weiteren Programmen und probieren Sie diese ebenfalls aus. Möglicherweise finden Sie auch Angebote, die für die Allgemeinheit zwar kostenpflichtig, aber für Studierende kostenlos sind.

- Installieren und ausführen:

  ```
  sudo apt install cppcheck
  cppcheck --enable=all --suppress=missingIncludeSystem main.c
  ```

- Installieren und ausführen:

  ```
  sudo apt install clang-tidy
  clang-tidy --checks='-*,clang-analyzer-*' main.c --
  ```

- Der Compiler `gcc` sollte bereits installiert sein. Die Option `-O3` aktiviert diverse Performance-Optimierungen, die unter anderem auch dazu führen, dass `gcc` Code genauer analysiert.

  ```
  gcc -O3 -Wall -Wextra main.c
  ```

Mit dem Skript `install.sh` können Sie alle Programme unter den meisten Debian-basierten Betriebssystemen installieren und mit `test.sh` können Sie sie ausführen.