Human-Computer Interaction Group

Leibniz Universität Hannover

Programmieren 1 – WS 2017/18

Prof. Dr. Michael Rohs, Oliver Beren Kaul, M.Sc., Tim Dünte, M.Sc.

Übungsblatt 9

Alle Übungen (bis auf die erste) müssen in Zweiergruppen bearbeitet werden. Beide Gruppenmitglieder müssen die Lösung der Zweiergruppe einzeln abgeben. Die Namen beider Gruppenmitglieder müssen sowohl in der PDF Abgabe als auch als Kommentar in jeglichen Quelltextabgaben genannt werden. Plagiate führen zum Ausschluss von der Veranstaltung.

Abgabe bis Donnerstag den 21.12. um 23:59 Uhr über https://assignments.hci.uni-hannover.de/WiSe2017/Programmieren1. Die Abgabe muss aus einer einzelnen zip-Datei bestehen, die den Quellcode, ein pdf bei Freitextaufgaben und alle weiteren nötigen Dateien (z.B. Eingabedaten oder Makefiles) enthält. Lösen Sie Umlaute in Dateinamen bitte auf.

Aufgabe 1: Cars

Das Template für diese Aufgabe ist cars.c. Eine Reihe von Autos soll hinsichtlich verschiedener Attribute verglichen werden. Ein Auto hat diese Attribute:

- `model` – Name (String)
- `power` - in kW (double)
- `weight` - in kg (double)
- `top_speed` - in km/h (double)
- `co2` - Emissionen in g/km (double)

Ein Auto wird durch das `struct Car` repräsentiert. Die Konstruktor-Funktion `new_car` allokiert ein `Car` auf dem Heap und initialisiert die Attribute. In der `main`-Funktion werden im Array `cars` 6 Autos eingetragen. In den folgenden Teilaufgaben sollen einige Funktionen nacheinander implementiert werden, die in der `main`-Funktion aufgerufen werden. Die Stellen, an denen Sie etwas implementieren sollen sind mit `todo` markiert.

a) Implementieren Sie die Konstruktor-Funktionen `make_stringarray` und `make_doublearray`. Orientieren Sie sich dazu an `make_cararray`. Die eigentlichen C-Arrays sollen jeweils dynamisch auf dem Heap allokiert werden. Verwenden Sie `xcalloc/xmalloc` und `free` zur dynamischen Anforderung bzw. Freigabe von Speicher.

b) Implementieren Sie die Funktion `car_models`, welche das `cars` Array entgegennimmt, und ein String Array zurückgibt, das alle Modellnamen der `cars` enthält. Hinweis: Es muss zuerst ein neues `StringArray` erstellt werden.

c) Implementieren Sie die Funktion `print_string_array`, das alle Elemente eines String Arrays nacheinander jeweils in einer neuen Zeile ausgibt.

d) Implementieren Sie die Funktion `average_co2_emissions`, die das `cars` Array entgegennimmt und die durchschnittlichen CO_2-Emissionen aller Elemente in `cars` berechnet.

e) Implementieren Sie die Funktion `power_weight_ratios`, welche das `cars` Array entgegennimmt und ein `DoubleArray` zurückgibt, das die `power-to-weight-ratio` (power / weight) für jedes Auto enthält.

f) Implementieren Sie die Funktion `power_at_most`, welche das `cars` Array und ein Limit für einen maximalen power Wert entgegennimmt. Als Rückgabewert soll diese Funktion ein neues Array aus genau denjenigen Car Elementen generieren, bei denen der power Wert nicht höher ist als das Limit. Das neue Array soll nur genau so groß sein, wie nötig. Das Eingabe-Array soll nicht verändert werden.

g) Geben Sie die dynamisch allokierten Daten am Ende der Funktion `cars_test` wieder frei. Achten Sie darauf, auch die einzelnen Elemente des `cars.a` Arrays freizugeben. Wenn allokierter Speicher nicht wieder freigegeben wird, erscheint beim Beenden des Programms eine Fehlermeldung:
`40 bytes allocated in new_car (cars.c at line 25) not freed`

Aufgabe 2: Matrizen

Das Template für diese Aufgabe ist matrix.c. In dieser Aufgabe geht es um dynamisch allokierte Matrizen. Eine Matrix wird durch einen Zeiger auf eine `struct Matrix` repräsentiert. Diese Struktur enthält die Angabe der Anzahl von Zeilen und Spalten sowie einen Zeiger auf ein C-Array, das Zeiger auf die Zeilen der Matrix enthält. Jede Zeile der Matrix ist ein C-Array mit Elementen vom Typ double.

Die Zeilen und Spalten der Matrix sollen dynamisch allokiert werden. Verwenden Sie `xcalloc/xmalloc` und `free` zur dynamischen Anforderung bzw. Freigabe von Speicher. Der dynamisch angeforderte Speicher soll vor dem Ende des Programms auch wieder freigegeben werden.

a) Implementieren Sie die Funktion `make_matrix`, die dynamisch eine Matrix mit der entsprechenden Anzahl Zeilen und Spalten erzeugt, die Elemente mit 0 initialisiert und einen Zeiger auf die erzeugte Matrix zurückgibt.

b) Implementieren Sie die Funktion `copy_matrix`. Diese Funktion bekommt einen Zeiger auf ein eindimensionales C-Array mit `n_rows * n_cols` double-Werten übergeben. Es soll nun eine Matrix dynamisch erzeugt werden und die übergebenen double-Werte in diese Matrix kopiert werden. Die Werte sind in der Eingabe zeilenweise angeordnet.

c) Implementieren Sie die Funktion `print_matrix`, die eine Matrix sinnvoll formatiert ausgibt.

d) Implementieren Sie die Funktion `transpose_matrix`, die als Eingabe einen Zeiger auf eine Matrix erhält und einen Zeiger auf eine neue transponierte Matrix zurückgibt. Die Eingabematrix darf nicht verändert werden.

e) Implementieren Sie die Funktion `mul_matrices`, die zwei Matrizen multipliziert. Die Eingabematrizen dürfen dabei nicht verändert werden, sondern das Ergebnis soll als neue Matrix dynamisch erzeugt und zurückgegeben werden. Die Funktion soll auch überprüfen, ob die Dimensionen der Argumente kompatibel sind. Geben Sie im Fehlerfall `NULL` zurück.

f) Implementieren Sie die Funktion `free_matrix`, die eine dynamisch allokierte Matrix freigibt. Wenn allokierter Speicher nicht wieder freigegeben wird, erscheint beim

Beenden des Programms folgende Fehlermeldung:
`4 bytes allocated in make_matrix (matrix.c at line 123) not freed`

Aufgabe 3: Reversi Zufallsspieler

In dieser Aufgabe soll das Reversi-Spiel weiterentwickelt werden. Diesmal soll ein Computer-Spieler entwickelt werden, der einen zufälligen gültigen Zug mach. Außerdem sollen einige Hilfsfunktionen implementiert werden.

a) Implementieren Sie die Funktion `print_position`, die Ausgaben der Form Buchstabe-Zahl – z.B. „D1" für die Position (3, 0) – erzeugt.

b) Auf dem Positions-Stack soll jedes Mal, wenn der Computer am Zug ist, alle in dieser Situation gültigen Züge gespeichert werden. Implementieren Sie die Funktionen `push` und `pop` des Positions-Stacks. Brechen Sie das Programm bei Stack-Überlauf oder Stack-Unterlauf ab.

c) Implementieren Sie die Funktion `random_position`. Diese soll eine zufällige Position vom Stack zurückgeben, ohne den Stack zu verändern. Verwenden Sie `i_rnd(n)`.

d) Implementieren Sie die Funktion `computer_move`. Diese soll für alle Positionen auf dem Spielbrett testen, ob der momentan zu setzende Stein („my_stone") dort gesetzt werden darf. Alle möglichen Positionen (also alle gültigen Züge) sollen auf dem Stack gespeichert werden. Zuletzt soll eine zufällige gültige Position zurückgegeben werden. Wenn kein gültiger Zug möglich ist, soll (-1, -1) zurückgegeben werden.

e) Modifizieren Sie `human_move` so, dass bei Eingabe von `?` in allen Feldern, in denen ein Stein („my_stone") gesetzt werden darf ein * erscheint. Danach ist der Spieler weiterhin am Zug und muss eine gültige Position eingeben. Zur Implementierung bietet es sich an, Hilfsfunktionen zu implementieren. Es folgt ein Beispiel:

```
  |A|B|C|D|E|F|G|H|
1|_|_|_|_|_|_|_|_|
2|_|_|_|_|_|_|_|_|
3|_|_|_|_|_|_|_|_|
4|_|_|X|X|X|_|_|_|
5|_|_|O|X|O|_|_|_|
6|_|_|_|O|_|_|_|_|
7|_|_|O|_|_|_|_|_|
8|_|_|_|_|_|_|_|_|
Score for O: 0
X's move: ?            ← Spieler gibt '?' ein
  |A|B|C|D|E|F|G|H|
1|_|_|_|_|_|_|_|_|
2|_|_|_|_|_|_|_|_|
3|_|_|_|_|_|_|_|_|
4|_|_|X|X|X|_|_|_|
5|_|*|O|X|O|*|_|_|    ← Das Spielbrett zeigt die möglichen Züge für 'X'
6|_|*|*|O|*|*|_|_|
7|_|_|O|*|_|_|_|_|
8|_|_|_|_|_|_|_|_|
```

Hinweise zum Editieren, Compilieren und Ausführen:

- mit Texteditor `file.c` editieren und speichern
- `make file` ← ausführbares Programm erstellen
- `./file` ← Programm starten (evtl. ohne `./`)
- Die letzten beiden Schritte lassen sich auf der Kommandozeile kombinieren zu:
  `make file && ./file`