Human-Computer Interaction Group

Leibniz Universität Hannover

Programmieren 1 – WS 2019/20

Prof. Dr. Michael Rohs, Tim Dünte, M.Sc.

Übungsblatt 11

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. Wenn Sie für Übungsblatt 1 noch keinen Gruppenpartner haben, geben Sie alleine ab und nutzen Sie das erste Tutorium dazu, mit Hilfe des Tutors einen Partner zu finden. Plagiate führen zum Ausschluss von der Veranstaltung.

Abgabe bis Donnerstag den 16.01. um 23:59 Uhr über https://assignments.hci.uni-hannover.de/WiSe2019/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.

Die Dokumentation der Prog1lib finden Sie unter: https://hci.uni-hannover.de/files/prog1lib/index.html

Aufgabe 1: Zeigerliste: Städte

Das Template für diese Aufgabe ist city_list.c. In dieser Aufgabe werden Operationen einer Zeigerliste verwendet. Die Zeigerliste ist in pointer_list.h bzw. pointer_list.c implementiert. Die Beispiele aus der Vorlesung finden sich in book_list.c. Machen Sie sich zunächst mit pointer_list und book_list vertraut und lesen Sie den Beispielcode gründlich. Implementieren Sie dann die unten angegebenen Operationen auf einer Liste von Städten. Geben Sie nicht mehr benötigten Speicher jeweils frei. Verwenden Sie für diese Aufgabe ausschließlich die Listenimplementierung in pointer_list.h bzw. pointer_list.c.

a) Implementieren Sie die Funktion `copy_city`, die eine übergebene City dupliziert. Verwenden Sie die Konstruktorfunktion `new_city`. Implementieren Sie ebenfalls die Funktion `free_city`, die von einer City belegten Speicherplatz freigibt.

b) Implementieren Sie die Funktion `small_city`, die genau dann wahr liefert, wenn die übergebene Stadt weniger als 100000 Einwohner hat. Die Funktion soll im Zusammenhang mit `find_list` verwendet werden (siehe Aufruf in der `main`-Funktion).

c) Erklären Sie als Kommentare im Quelltext jede Zeile der Funktion `area_less_than`. Implementieren Sie in der `main`-Funktion einen geeigneten Aufruf der `find_list`-Funktion, so dass die erste Stadt mit einer Fläche kleiner als `max_area` zurückgeliefert wird und geben Sie diese aus.

d) Implementieren Sie die Funktion `city_name`, die den Namen der übergebenen Stadt extrahiert. Die Funktion soll im Zusammenhang mit `map_list` verwendet werden (siehe Aufruf in der `main`-Funktion).

Aufgabe 2: Zeigerliste: Städte (Fortsetzung)

e) Implementieren Sie die Funktion `multiply_inhabitants`, die im Zusammenhang mit `map_list` verwendet werden soll (siehe Aufruf in der `main`-Funktion). Die zu implementierende Funktion soll eine Kopie der in `element` übergebenen Stadt erstellen, die Einwohnerzahl in der Kopie mit dem übergebenen Faktor multiplizieren und die so veränderte Kopie zurückgeben.

f) Erstellen Sie in der main-Funktion unter Benutzung von `filter_list` eine Liste aller Kleinstädte (unter 100000 Einwohnern) und geben diese aus. Erstellen Sie außerdem mit `filter_map_list` eine Liste der Namen aller Kleinstädte und geben diese aus. Geben Sie nach der Ausgabe den Speicher für diese Listen wieder frei.

g) Erklären Sie als Kommentare im Quelltext jede Zeile der Funktionen `add_area` und `larger_city`, die im Zusammenhang mit `reduce_list` verwendet werden (siehe Aufrufe in der `main`-Funktion).

h) Sortieren Sie die Städte in `list` aufsteigend nach Einwohnerzahl durch sortiertes Einfügen (`insert_ordered`). Der entsprechende Aufruf ist in der `main`-Funktion zu finden. Implementieren Sie dazu die Vergleichsfunktion `cmp_city_inhabitants`.

Aufgabe 3: Zeigerliste erweitern

Das Template für diese Aufgabe ist pointer_list_ext.c. In dieser Aufgabe soll die Zeigerliste um weitere Operationen erweitert werden. Geben Sie nicht mehr benötigten Speicher jeweils frei. Verwenden Sie für diese Aufgabe ausschließlich die Listenimplementierung in pointer_list.h bzw. pointer_list.c.

a) Implementieren Sie die Funktion `take_list`, die eine neue Liste mit den ersten `n` Elementen der Eingabeliste erzeugt. Die Elemente selbst sollen nicht dupliziert werden.

b) Implementieren Sie die Funktion `drop_list`, die eine neue Liste mit den Elementen der Eingabeliste außer den ersten `n` Elementen erzeugt. Die ersten Elemente der Eingabeliste werden also nicht verwendet. Die Elemente selbst sollen nicht dupliziert werden.

c) Implementieren Sie die Funktion `interleave`, die eine neue Liste erzeugt, indem sie, beginnend bei Liste1, abwechselnd ein Element aus Liste1 und Liste2 nimmt. Die Eingabelisten sollen nicht veränert werden. Die Elemente selbst sollen nicht dupliziert werden.

d) Implementieren Sie die Funktion `group_list`, die äquivalente Elemente der Eingabeliste gruppieren soll. Für äquivalente Elemente liefert die Funktion `equivalent` den Wert wahr. Für die Beispielfunktion `group_by_length` sind Strings äquivalent, wenn sie die gleiche Länge haben. Das Resultat von `group_list` ist eine Liste von Gruppen. Jede Gruppe ist eine Liste von äquivalenten Elementen. Beispiel: Für die Liste `["x", "yy", "zzz", "f", "gg"]` ergeben sich nach Aufruf von `group_list(list, group_by_length)` die Gruppen `[["x", "f"], ["yy", "gg"], ["zzz"]]`. Die Reihenfolge der Gruppen und der Elemente in den Gruppen ist dabei beliebig.

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`