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 8

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 14.12. 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: Sortieren durch zufälliges Vertauschen

Implementieren Sie aufbauend auf dem Template `random_sort.c` einen Sortieralgorithmus, der durch zufälliges Tauschen von zwei Elementen eines Arrays dieses sortiert. Es sollen Autos sortiert werden, die von einer Konstruktorfunktion erzeugt werden.

a) Schreiben Sie eine Funktion `int compare(Car car1, Car car2)`, die zwei Autos vergleicht und 1 zurückgibt, wenn car1 jünger ist als car2 und -1, wenn car1 älter ist als car2. Wenn beide Autos gleich alt sind, dann soll die Funktion auch die Marke überprüfen und diese lexikographisch sortieren. Sind zwei Autos gleich alt und haben die gleiche Marke, so soll 0 zurückgegeben werden. Nutzen Sie für den Vergleich von zwei Zeichenketten die Funktion `int strcmp(char* str1, char* str2)`. Diese liefert -1, 0 oder 1 zurück, abhängig davon ob `str1` lexikographisch kleiner, gleich oder größer ist als `str2`.

b) Schreiben Sie eine Testfunktion `void compare_test(void)`, mit mindestens 5 Testfällen, die die korrekte Funktion Ihrer `compare` Funktion überprüft.

c) Implementieren Sie eine Funktion `bool sorted(Car* a, int length)`, die testet, ob das Array sortiert ist. Nutzen Sie die bereits implementierte `compare` Funktion. Die Funktion soll `true` zurückgeben, wenn das Array sortiert ist, ansonsten `false`.

d) Implementieren Sie die Funktion `int random_sort(Car* a, int length)`, die solange zwei zufällige Autos in dem Array vertauscht, bis die Liste sortiert ist. Nutzen Sie die Funktion `sorted` nach jedem Tausch, um zu testen ob, das Array nun sortiert ist. Die Funktion soll zunächst nur 0 zurückgeben, später (e) soll die Anzahl der Vertauschungen zurückgegeben werden.

e) Überprüfen Sie, wie oft Ihre Funktion Elemente vertauscht. Je öfter dies passiert, desto ineffizienter ist die Funktion. Fügen Sie dazu eine Zählvariable `swaps` in Ihre `random_sort` Funktion ein, die jedes Mal inkrementiert wird, wenn ein Tausch stattfindet. Nach der Sortierung soll `swaps` zurückgeben werden. Können Sie über diese Variable die Anzahl der Aufrufe der `compare` Funktion bestimmen? Wenn ja, wie hängt diese von `swaps` ab?

f) Testen Sie Ihren Algorithmus für mindestens 5 verschieden große zufällige Arrays jeweils 100-mal (Arraygrößen im Bereich von 3 – 10). Geben Sie jeweils den Durchschnittswert von `swaps` für die verschiedenen Arraylängen auf der Konsole aus, sowie die Anzahl der stattgefundenen Aufrufe von `compare` (falls möglich). Ein Array mit beliebiger Länge und zufälligen Autos können Sie mit der `create_car_park(int car_count)` Methode erstellen.

## Aufgabe 2: Array Merge

In dieser Aufgabe sollen die Daten aus zwei sortierten Arrays in ein Array sortiert eingefügt werden. Diese Funktionalität ist bekannt aus dem Mergesort Algorithmus. In dieser Aufgabe sollen jedoch Fahrräder nach ihrer „Coolness" sortiert werden.

a) Vervollständigen Sie die Methode `double calculate_coolness(Bike* bike)`, die die „Coolness" eines Fahrrads berechnet. Für die Berechnung wird der Preis, die Anzahl an Gängen und die Reifengröße herangezogen. Für jeden einzelnen Wert wird ein Faktor berechnet. Alle 3 Faktoren multipliziert ergeben die Coolness des Fahrrads.

- Für den Preisfaktor gilt: $-2.75\times10^{-6}\times(preis-1000)^2+3$.
- Für den Gangschaltungsfaktor gilt: $\frac{1}{7}\times gangzahl+0.5$.
- Für den Reifengrößenfaktor gilt: Liegt die Reifengröße des Fahrrads im Intervall (24, 29) beträgt der Faktor 1.9, für alle anderen Größen ist der Faktor 0.9.

b) Vervollständigen Sie die Funktion `int compare(Bike* bike1, Bike* bike2)`. Diese soll die Coolness von zwei Fahrrädern vergleichen. Wenn der Wert von `bike1` kleiner als von `bike2` ist soll -1 zurückgegeben werden, ist es andersherum, so soll 1 zurückgegeben werden. Haben beide Fahrräder die gleiche Coolness so soll 0 zurückgegeben werden.

c) Vervollständigen Sie die Funktion `void merge_arrays(Bike* bikes1, int n1, Bike* bikes2, int n2, Bike* bout)`. Die Arrays `bikes1` mit Länge `n1` und `bikes2` mit Länge `n2` enthalten jeweils schon nach Coolness sortierte Fahrräder. Nach dem Aufruf dieser Funktion sollen in bout sortiert die Fahrräder aus beiden Arrays stehen. Nutzen Sie dafür die Methode des sortierten Einfügens bekannt aus dem Mergesort Algorithmus. In der vorgegeben Funktion `merge_arrays_test` werden abschließend Arrays mit sortierten Fahrrädern erzeugt, die `merge_arrays` Funktion wird aufgerufen und am Ende wird getestet ob das resultierende Array korrekt sortiert ist.

## Aufgabe 3: Reversi

In dieser Aufgabe geht es darum, das Spiel Reversi zu implementieren, so dass von menschlichen Spielern Züge eingegeben werden können. Reversi ist ein Brettspiel für zwei Spieler, das auf einem Spielbrett mit 8x8 Feldern gespielt wird. Die verwendeten Spielsteine haben zwei unterschiedliche Seiten ('X' für Spieler 1 und 'O' für Spieler 2). Die Grundregeln lauten:

- 'X' beginnt.

- Horizontal, vertikal oder diagonal in einer Reihe liegende Steine einer Farbe werden umgedreht, wenn sie am Anfang und Ende von gegnerischen Steinen eingerahmt sind.
- Ein Zug ist nur dann gültig, wenn er zum Umdrehen mindestens eines gegnerischen Steins führt.
- Wenn keiner der Spieler einen gültigen Zug machen kann, endet das Spiel. Der Spieler mit den meisten Steinen hat gewonnen. Bei Gleichstand endet das Spiel unentschieden.

Details finden sich unter: https://de.wikipedia.org/wiki/Othello_(Spiel)

Hier ein eine beispielhafte Programmausgabe an Hand einiger Züge.

```
  |A|B|C|D|E|F|G|H|
1|_|_|_|_|_|_|_|_|
2|_|_|_|_|_|_|_|_|
3|_|_|_|_|_|_|_|_|
4|_|_|_|O|X|_|_|_|
5|_|_|_|X|O|_|_|_|
6|_|_|_|_|_|_|_|_|
7|_|_|_|_|_|_|_|_|
8|_|_|_|_|_|_|_|_|
X's turn: D3          ← Eingabe Spieler X: D3
  |A|B|C|D|E|F|G|H|
1|_|_|_|_|_|_|_|_|
2|_|_|_|_|_|_|_|_|
3|_|_|_|X|_|_|_|_|
4|_|_|_|X|X|_|_|_|
5|_|_|_|X|O|_|_|_|
6|_|_|_|_|_|_|_|_|
7|_|_|_|_|_|_|_|_|
8|_|_|_|_|_|_|_|_|
Score for X: 3
O's turn: c5          ← Eingabe Spieler O: C5
  |A|B|C|D|E|F|G|H|
1|_|_|_|_|_|_|_|_|
2|_|_|_|_|_|_|_|_|
3|_|_|_|X|_|_|_|_|
4|_|_|_|X|X|_|_|_|
5|_|_|O|O|O|_|_|_|
6|_|_|_|_|_|_|_|_|
7|_|_|_|_|_|_|_|_|
8|_|_|_|_|_|_|_|_|
```

```
Score for O: 0
X's turn: b6          ← Eingabe Spieler X: B6
  |A|B|C|D|E|F|G|H|
1|_|_|_|_|_|_|_|_|
2|_|_|_|_|_|_|_|_|
3|_|_|_|X|_|_|_|_|
4|_|_|_|X|X|_|_|_|
5|_|_|X|O|O|_|_|_|
6|_|X|_|_|_|_|_|_|
7|_|_|_|_|_|_|_|_|
8|_|_|_|_|_|_|_|_|
Score for X: 3
```

Das Template für diese Aufgabe ist reversi.c. Zunächst soll nur nach jedem Zug durch einen Spieler das Spielbrett dargestellt werden. Das Programm soll jeweils prüfen, ob der Zug korrekt ist und die betroffenen Steine umdrehen. Das Programm soll in den nächsten Übungen weiter ausgebaut werden.

a) Implementieren Sie die Funktion Game `init_game(char my_stone)`, so dass die Anfangsaufstellung erzeugt und das übergebene Zeichen (`'X'` oder `'O'`) als eigener Stein gespeichert wird.

b) Implementieren Sie die Funktion `print_board`, so dass der Spielzustand in der oben gezeigten Form ausgegeben wird.

c) Implementieren Sie die Funktionen `legal_dir` und `legal`. Diese sollen prüfen, ob ein Stein der Farbe `my_stone` an Position (x, y) gesetzt werden darf. Dies ist dann der Fall, wenn das Feld noch leer ist, die Position sich in den Grenzen des Spielbretts befindet und in mindestens einer Richtung (`direction`) gegnerische Steine umgedreht werden können. Die Funktionen `legal_dir` und `legal` sollen den Spielzustand nicht ändern.

d) Implementieren Sie die Funktionen `reverse_dir` und `reverse`. Diese sollen einen eigenen Stein (`my_stone`) auf Position (x,y) setzen und alle dadurch eingerahmten gegnerischen Steine umdrehen. Wenn das Setzen an Position (x,y) kein legaler Zug wäre, soll nicht gesetzt werden.

e) Implementieren Sie die Funktion `count_stones`, die die Anzahl der Steine einer bestimmten Farbe auf dem Spielbrett zählt.

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`