Human-Computer Interaction Group

Leibniz Universität Hannover

Programmieren 1 – WS 2019/20

Prof. Dr. Michael Rohs, 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. 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.

Ausnahmsweise Abgabe bis Freitag den 13.12. um 23:59 Uhr (wie in der Übung besprochen) 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: Memory

In dieser Aufgabe geht es darum, das Spiel „Memory" zu implementieren, so dass man als Einzelspieler auf der Konsole spielen kann. Es geht darum, Paare gleicher, verdeckt liegender Karten durch Aufdecken von jeweils zwei Karten zu finden. Eine detailliertere Spielbeschreibung findet sich z.B. unter https://de.wikipedia.org/wiki/Memory_(Spiel). In der zu implementierenden Variante soll nur ein einzelner Spieler spielen können. Die Versuche und Punkte sollen vom Computer gezählt werden. Die zugehörige Template-Datei ist `memory_game.c`. Hier eine Beispielausgabe:

```
  1 2 3
1 # # #
2 # # #                    ← Anfangszustand mit sechs verdeckten Karten
0 points, 0 turns
> 11                       ← Eingabe: Spieler möchte Zeile 1, Spalte 1 aufdecken

  1 2 3
1 B # #                    ← Karte in Zeile 1, Spalte 1 aufgedeckt
2 # # #
0 points, 1 turns
> 12                       ← Eingabe: Spieler möchte Zeile 1, Spalte 2 aufdecken

  1 2 3
1 B C #                    ← zwei aufgedeckte Karten
2 # # #
> 0 points, 2 turns
```

```
press <return> to continue
```
← warten auf Eingabe, da B≠C wird wieder verdeckt

a) Die Funktion `shuffle` mischt die Karten (Einträge des Arrays a der Länge n) zufällig. Erläutern Sie wie die Arbeitsweise von `shuffle`. Angenommen der Ausdruck `rand() % (i + 1)` liefere gleichverteilt zufällige ganze Zahlen im Intervall [0, i]. Sind dann alle möglichen Permutationen des Arrays gleich wahrscheinlich?

b) Implementieren Sie die Funktion `init_cards`. Nehmen Sie an, dass das übergebene Array bereits existiert und dass lediglich die Zeichen im Array `cards` so gesetzt werden müssen, dass jedes verwendete Symbol genau zweimal vorkommt und dass die Anordnung zufällig ist. Nicht verwendete Plätze auf dem Spielfeld sollen mit Leerzeichen aufgefüllt werden. Als Symbole sollen die ASCII-Zeichen ab `'A'` verwendet werden. Aufgedeckte Karten werden über ihren den ASCII-Wert (z.B. `'A'` = 65), verdeckte Karten über den negativen ASCII-Wert des Symbols (z.B. `-'A'` = -65) repräsentiert.

c) Implementieren Sie die Funktion `print_board`, die den aktuellen Zustand des Spielfelds ausgibt. Die Ausgaben sollen aussehen, wie in der obigen Beispielausgabe und wie in den Kommentaren im Quelltext beschrieben. Die Zeichen sind im `cards`-Array zeilenweise von oben nach unten abgelegt.

d) Implementieren Sie die Funktion `int array_index(Board *b, int r, int c)`, die den zu den Koordinaten (row, column) gehörigen Array-Index in `cards` berechnet. Die Zeichen sind im `cards`-Array zeilenweise von oben nach unten abgelegt. Die Koordinaten (0,0) entsprechen also Index 0, die Koordinaten (0,1) dem Index 1 und die Koordinaten (1,0) dem Index `b->cols`. Das Programm muss mit `exit(1);` beendet werden, wenn ungültige Spalten- oder Zeilenwerte übergeben wurden.

e) Implementieren Sie unter Verwendung von `array_index` die Funktionen `get`, `set` und `turn` zum Lesen, Setzen bzw. Umdrehen einer Karte an der Position (row, column).

f) Implementieren Sie die Funktion `int clamp(int x, int low, int high)`, die den Wert von x auf das Intervall [low, high] beschränkt.

g) Erklären Sie den Aufbau der Funktion `do_move`. Ist die Funktion robust gegenüber Fehleingaben oder kann das Programm durch bestimmte Eingaben in einen inkonsistenten Zustand gebracht werden? Begründen Sie Ihre Antwort.

Stellen Sie sicher, dass Ihre Lösungen die in der Testfunktion `tests` aufgeführten Testfälle erfüllen.

Aufgabe 2: Operationen auf Arrays

In dieser Aufgabe sollen Sie verschiedene Operationen für Integer Arrays implementieren. Das Template für diese Aufgabe ist `array_operations.c`. Testfälle für die verschiedenen Aufgabenteile sind bereits vorhanden. Ergänzen Sie diese, wenn Sie das für nötig halten.

a) Implementieren Sie als erstes die Funktion `bool compare(int *array_a, int length_a, int *array_b, int length_b)`. Diese soll zwei Arrays `array_a` und `array_b` vergleichen und `true` zurückgeben, wenn der Inhalt beider Arrays identisch ist. Andernfalls soll `false` zurückgegeben werden. Diese Funktion ist relevant für die weiteren Teilaufgaben, da sie auch in den Testfällen der späteren Teilaufgaben

verwendet wird. Bearbeiten Sie die weiteren Teilaufgaben erst, wenn Sie sicher sind, dass compare funktioniert (Alle Testfälle werden erfüllt).

b) Implementieren Sie die Funktion `int remove_negatives(int* array, int length)`, die ein Array übergeben bekommt und in diesem in-place die negativen Zahlen entfernt. Dafür werden die verbleibenden positiven Zahlen an den Anfang des Arrays verschoben und die Anzahl an positiven Elementen als neue Länge zurückgegeben. D.h. Sie allokieren keinen neuen Speicher, sondern verändern das bestehende Array. Beispiele:

{1, -2, 3} -> {1, 3}
{-1, 2, 2} -> {2, 2}

c) Implementieren Sie die Funktion `int remove_even_numbers(int *array, int length)`, die ein Array übergeben bekommt und in diesem in-place alle geraden Zahlen entfernt. Dafür werden die verbleibenden ungeraden Zahlen an den Anfang des Arrays verschoben und die Anzahl an ungeraden Elementen als neue Länge zurückgegeben. D.h. Sie allokieren keinen neuen Speicher, sondern verändern das bestehende Array. Beispiele:

{1, -2, 3} -> {1, 3}
{-1, 2, 2} -> {-1}

d) Machen Sie sich mit der Struktur `Better_Array` vertraut und implementieren Sie danach die Funktion `Better_Array intersect(int *array_a, int length_a, int* array_b, int length_b)`. Diese bekommt zwei Arrays übergeben und bildet die Schnittmenge aus beiden. Die Schnittmenge wird auch hier inplace in einem der übergebenen Arrays gespeichert. Wählen Sie dazu das Array aus, dass eine kleinere Länge hat. Sollten beide Arrays gleich lang sein, so überschreiben Sie array_a. Geben Sie dann eine entsprechend initialisierte Struktur Better_Array mit dem Ergebnis und der Länge des Ergebnisses zurück. Beispiele:

{1} und {1, 2, 3, 4} -> {1}
{1, 4, 3} und {1, 2, 3, 4} -> {1, 4, 3}

e) Implementieren Sie die Funktion void `merge_sorted_arrays(int *array_a, int length_a, int* array_b, int length_b, int* result, int length_result)`, die zwei sortierte Arrays (array_a und array_b) zu einem sortierten Array `result` zusammenfügt. Beispiele:

{1, 2, 15} und {3, 4, 17} -> {1, 2, 3, 4, 15, 17}
{1} und {1, 2, 3, 4} -> {1, 1, 2, 3, 4}

f) Welche Vorteile bietet die Struktur `Better_Array`? Wie werden Strings in C repräsentiert? Wie ist in C die Länge eines Strings gespeichert bzw. wie wird sie bestimmt? Ist so etwas auch für int[] möglich? Beantworten Sie diese Fragen als Kommentar in Ihrer Quelltextdatei.

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 (`dir`ection) 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`