Einführung in die Programmierung
WS 2015/2016
Blatt 10

Simon Hangl, Sebastian Stabinger, Benedikt Hupfauf, Johannes Kessler

2016–01–13

- Abgabe bis spätestens Dienstag 21:59:59 über OLAT (`https://lms.uibk.ac.at/olat/dmz/`).
- Bereiten Sie jede Aufgabe so vor, dass Sie Ihre Lösung im Proseminar präsentieren können!
- Benennen Sie Ihre Abgabe nach folgendem Schema: *Gruppennummer-Nachname*`-blatt`*Übungsblattnummer*`.tar.gz` Wenn Sie also Max Mustermann heißen und Gruppe 1 besuchen, heißt die Datei von Übung 10: `1-mustermann-blatt10.tar.gz`
- Compilieren Sie alle Programme mit den Optionen `-Wall -Werror -std=c99`

Feedback

Nutzen Sie die angebotenen Möglichkeiten, uns Feedback zu geben (eMail, Tutorium, Proseminar). Hier können Sie uns auf Probleme, notwendige Stoffwiederholungen, Unklarheiten, aber auch positive Dinge, die beibehalten werden sollten, hinweisen.

Testen und Dokumentation

Stellen Sie sicher, dass alle Lösungen fehlerfrei kompilieren. Testen Sie Ihre Lösungen ausführlich (z.B. auf falsche Eingaben, falsche Berechnungen, Sonderfälle) und dokumentieren Sie sie. Dies hilft Ihnen bei der Präsentation und uns beim Nachvollziehen Ihrer Entscheidungen. Lesen Sie die Aufgaben *vollständig* durch.

Aufgabe 1 (2 Punkte)

- Schreiben Sie eine Funktion `map` welcher ein Array vom Typ `int`, die Länge des Arrays und ein Funktionspointer übergeben wird. Der Funkti-

onspointer soll auf eine Funktion verweisen welche ein `int` als Parameter übernimmt und `int` zurück liefert.

Ihre Funktion `map` soll für jedes Element des Arrays die übergebene Funktion aufrufen und den Rückgabewert statt des originalen Elements im Array speichern.

- Schreiben Sie drei Funktionen welche:
  - die übergebene Zahl mit `2` multipliziert
  - die übergebene Zahl quadriert
  - die Wurzel der übergebenen Zahl berechnet
- Schreiben Sie ein kurzes Testprogramm welches diese drei Funktionen mittels `map` auf ein Array anwendet und geben Sie die Ergebnisse aus.

Beispiel: Ausgehend von einem Array mit den Werten `1,2,3,4,5` würde nach Aufruf der `map` Funktion und übergabe der Quadrierfunktion als Funktionspointer folgendes im Array stehen: `1,4,9,16,25`.

Hinweis: *Abgabe: 1-mustermann-a1.c*

Aufgabe 2 (2 Punkte)

Implementierten Sie die Caesar Verschlüsselung für Kleinbuchstaben.

Dabei wird jedes Zeichen c einzeln verschlüsselt, indem der gewählte numerische Schlüssel K addiert wird und mittels Modulo wieder ein Kleinbuchstabe erzeugt wird: $verschluesseln(S, c) = (c + K)\ mod\ 26$

Zum entschlüssen kann einfach ein negativer Schlüssel verwendet werden.

Ihr Programm soll mit zwei Argumenten aufrufbar sein: Das erste Argument ist der numerische Schlüssel (positiv oder negativ), das zweite Argument ist der Text der verschlüsselt werden soll.

Zum Beispiel:

```
./caesar 2 abcd
cdef

./caesar -2 cdef
abcd
```

Da die Argumente in `argv` als Zeichenketten vorliegen muss die Zahl mittels `man 3 strtol` oder `man 3 strtol` in eine ganze Zahl umgewandelt werden.

Das Programm soll alle Eingaben überprüfen und soll den verschlüsselten Text ausgeben. Im Fehlerfall soll das Programm nach Ausgabe einer entsprechenden Fehlermeldung beendet werden.

Zur Vereinfachung sind hier einige bash-Befehle mit dem Sie einen Teil der Funktionalität überprüfen können:

```
# testet verschlüssen und entschlüsseln
test ($(./caesar 100 "test" | xargs ./caesar -100) = "test") && echo \
richtig || echo falsch

# testet fehlerhaften Key
./caesar x test && echo "fehler nicht erkannt" || echo "fehler erkannt"

# testet fehlerhaften Text
./caesar x abCd && echo "fehler nicht erkannt" || echo "fehler erkannt"
```

Welche mögliche Falscheingabe wurde hier noch nicht getestet?

Hinweis: *Abgabe: 1-mustermann-a2.c*

Aufgabe 3 (5 Punkte)

Schreiben Sie Ihre Queue von Blatt 9 so um, dass sich das Verhalten wie folgt ändert:

1. Erzeugen Sie Ihr Array nicht statisch, sondern dynamisch (siehe `malloc`, `realloc`)
2. Sobald die Queue voll ist und ein weiteres Element eingefügt werden soll, soll sich die Größe des Arrays verdoppeln um Platz für neue Elemente zu schaffen
3. Wird ein Element entfernt und ist anschließend nur noch weniger als die Hälfte des Arrays belegt, soll das Array auf die Hälfte schrumpfen. Es soll aber immer mindestens Platz für 10 Elemente sein (d.h. das Array wird nicht auf weniger als 10 Elemente verkleinert).
4. Schreiben Sie Testfunktionen, die diese neue Funktionalität sinnvoll testen (z.b. Elemente so einfügen und rausnehmen, damit die Veränderungen der Arraygröße häufig auftreten).

Hinweis: *Abgabe: 1-mustermann-a3.c*

Aufgabe 4 (7 Punkte)

Implementieren Sie *Conway's Game of Life*[1][2].

Lassen Sie sich nicht von der langen Erklärung abschrecken. Die Implementierung ist nicht einmal 140 Zeilen lang.

Conway's Game of Life ist ein einfacher *zweidimensionaler* zellulärer Automat. Das quadratisch gerasterte Spielfeld hat eine bestimmte maximale Höhe und

[1] Wikipedia - Conway's Game of Life (en)
[2] Wikipedia - Conways Spiel des Lebens (de)

Breite. Jede Zelle dieses Spielfeldes kann einen von zwei möglichen Zuständen einnehmen und ist enweder *tot* oder *lebendig* und wird von genau 8 Nachbarn umschlossen.

Die Simulation wird schrittweise ausgeführt, wobei aus dem Spielfeld des vorhergegangen Schrittes ein neues berechnet wird. Für jede Zelle des alten Feldes wird ermittelt, welchen Zustand die Zelle an der selben Position im neuen Spielfeld einnehmen soll.

Dabei werden die folgenden vier Regeln verwendet:

- Eine tote Zelle mit genau drei lebenden Nachbarn wird in der Folgegeneration neu geboren.
- Lebende Zellen mit weniger als zwei lebenden Nachbarn sterben in der Folgegeneration an Einsamkeit.
- Eine lebende Zelle mit zwei oder drei lebenden Nachbarn bleibt in der Folgegeneration am Leben.
- Lebende Zellen mit mehr als drei lebenden Nachbarn sterben in der Folgegeneration an Überbevölkerung.

Wenn Sie das Spiel gerne vom Erfinder persönlich erklärt bekommen würden, können Sie auch das folgende (unterhaltsame) Video anschauen: Youtube - Does John Conway hate his Game of Life?.

Lösen Sie die Aufgabe indem Sie mit der vorgegebenen Datei `game_of_life.c` starten und wie folgt erweitern:

- Die Größe des Spielfeldes ist *konstant* und soll 160x40 Zellen betragen.
- Schreiben Sie eine Funktion welche ein Spielfeld initialisiert und Zufällig mit lebenden und toten Zellen befüllt. Es sollen dabei ca 20% der Zellen mit *lebend* initialisiert werden. Das Spielfeld soll dabei als Referenz übergegeben werden.
- Schreiben Sie eine Funktion welche die Anzahl der Nachbarn für eine bestimmte Zelle zurück gibt. Dazu werden die Koordinaten der Zelle und eine Referenz auf das Spielfeld übergeben.
- Schreiben Sie eine Funktion welche mit Hilfe der Anzahl an Nachbarn entscheidet welchen Zustand eine bestimmte Zelle in der nächsten Iteration hat. Dazu werden die Koordinaten der Zelle und eine Referenz auf das Spielfeld übergeben.
- Schreiben Sie eine Funktion welche einen einzelnen Spielschritt berechnet. Dazu wird eine Referenz auf das alte und das neue Spielfeld benötigt.
- Schreiben Sie eine Funktion die ein Spielfeld auf dem Terminal ausgibt. Dazu wird eine Referenz ein Spielfeld benötigt. Für alle toten Zellen soll ein Leerzeichen ausgegeben werden, für alle lebenden eine Raute (`a`). Es soll dabei nur jener Teil des Spielfeldes ausgegeben werden der im Terminal Platz hat. Dazu stehen die Funktionen `get_current_terminal_rows()` und `get_current_terminal_columns()` bereit die unter Linux die Dimensionen des Terminals zurückgeben.

- In der main-Funktion können/sollen zwei Spielfelder angelegt und (wenn nötig) initialisiert werden. Diese zwei Spielfelder sollen immer abwechslungsweise als neues- und altes Spielfeld verwendet werden. Das Programm soll nun die Spielschritte in einer Schleife ausführen bis das Programm beendet wird. Dabei soll das neue Spielfeld berechnet und ausgegeben werden. Vor der Ausgabe des Spielfeldes soll das Terminal mit `clear_terminal()` gelöscht werden. Wenn Sie nun zwischen jedem Durchlauf das Programm kurz pausieren (mit `man 3 sleep` oder `man 3 usleep`), bekommen Sie eine schöne Animation.

Das Problem, dass die Zellen am Spielfeldrand weniger als 8 Nachbarn haben kann wie folgt gelöst werden:

- Man behandelt diese Randfälle extra und zählt nur die wirklich gültigen Nachbarn.
- Man vergrößert die Spielfläche und lässt einen Rand von einer Zelle Breite.
- Das Spielfeld wird (logisch) zu einem Torus verbunden. Dabei bildet die Zellen des rechten Randes die linken Nachbarn des linken Randes und umgekehrt (damit hätte man einen Zylinder). Genauso wird die obere Kante mit der unteren verbunden um einen Torus zu bilden. Diese Lösung klingt kompliziert ist aber am einfachsten umzusetzen.

Wenn Sie alles richtig implementiert haben, können sich die folgenden Muster bilden: Wikipedia - Conway's Game of Life - Examples of patterns.

Hinweis: *Abgabe: 1-mustermann-a4.c*

Aufgabe 5 (3 Punkte)

Sortieren Sie ein `int`-Array mit der Funktion `qsort_r` (siehe man-pages!). Welche Bedeutung haben die einzelnen Parameter? Wofür werden diese benötigt?

Implementieren Sie folgende Ordnungen:

- Sortiert nach dem Wert der Zahl. Beispiel: 37 ist größer als 4, daher sollte 37 nach dem Sortieren hinter 4 kommen.
- Sortiert nach der Anzahl der Einsen in Binärdarstellung. Beispiel (für `char`): 57_{10} wird binär als $0011'1001_2$ dargestellt, hat also 4 Einsen. 7_{10} entspricht $0000'0111_2$, hat also nur 3 Einsen. Demnach sollte 7 nach dem Sortieren vor 57 sein.
- Sortiert nach der Länge der längsten Folge gleichwertiger Bits. Beispiel (für `char`): 13_{10} wird binär geschrieben als $0000'1101_2$. Die Länge der längsten Folge gleichwertiger Bits ist somit 4 (die vier Nullen am Anfang). 46_{10} enspricht binär $0010'1110_2$, daher ist die Länge der längsten Folge gleichwertiger Bits gleich 3 (die Einsen). Folglich sollte im sortierten Array 46 vor 13 stehen.

Implementieren Sie für jede Funktion die auf- und absteigende Ordnung! Wie können Sie das realisieren ohne zwei Versionen jeder Funktion zu implementieren?

Hinweis: *Abgabe: 1-mustermann-a5.c*