Human-Computer Interaction Group

Leibniz Universität Hannover

Übungsblatt 11

Programmieren 1 – WiSe 22/23

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

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 12.01. um 23:59 Uhr über https://assignments.hci.uni-hannover.de/WiSe2022/Prog1. Die Abgabe muss aus einer einzelnen Zip-Datei bestehen, die den Quellcode, ein PDF für Freitextaufgaben und alle weiteren nötigen Dateien (z.B. Eingabedaten oder Makefiles) enthält. Lösen Sie Umlaute in Dateinamen auf.

Hinweis: prog1lib

Die Dokumentation der prog1lib finden Sie unter der Adresse:
https://postfix.hci.uni-hannover.de/files/prog1lib/

Aufgabe 1: Öffnende und schließende Klammern

In dieser Aufgabe geht es um die Implementierung einer Syntaxprüfung von öffnenden und schließenden Klammern. Zu jeder öffnenden Klammer muss in der Zeichenkette ein passendes schließendes Gegenstück vorkommen. Schließende Klammern müssen zu der jeweiligen öffnenden Klammer passen. Folgende Klammernpaare sollen unterstützt werden: `( )`, `[ ]`, `{ }`, `< >`. Die Zeichenkette „`<{[()]}>`" ist valide. Die Zeichenketten „`(Test`" und „`([)]`" nicht.

Die Template-Datei für diese Aufgabe ist `parentheses.c`. Bearbeiten Sie alle mit `TODO` markierten Stellen.

a) Zum Lösen dieser Aufgabe ist es hilfreich einen Stack zu nutzen. Implementieren Sie diese Datenstruktur mit der für diese Aufgabe benötigten Funktionalität. Allokieren Sie Speicher dynamisch.

b) Implementieren Sie die Funktion `bool verify_parentheses(String text)`, die `true` zurück gibt wenn die Klammerung syntaktisch korrekt ist und `false` wenn nicht. Die Funktion muss in der Lage sein Zeichenketten beliebiger Länge zu behandeln.

c) Stellen Sie durch `report_memory_leaks(true)` sicher, dass dynamisch allokierter Speicher wieder freigegeben wird.

Aufgabe 2: Flattersatz

In dieser Aufgabe geht es um die Formatierung von Text auf eine vorgegebene Breite. Der Text soll im linksbündigen Flattersatz (https://de.wikipedia.org/wiki/Flattersatz) ausgegeben werden. Das erste Wort einer Zeile wird immer ausgegeben, auch wenn es breiter ist als die geforderte Breite. Das gewünschte Ergebnis für Breite 40 ist in **mytext_formatted.txt** zu sehen.

Die Lösung soll den Datentyp Str verwenden, der in der Template-Datei definiert wird. Dieser ermöglicht es, auf Teile von C-Strings zuzugreifen. Ein Str wird also nicht durch ein Nullbyte terminiert, sondern speichert explizit die Länge. Außerdem wird eine Kapazität gespeichert. Dies ermöglicht es, einen Puffer einer benötigten Kapazität zu allokieren und dann nach und nach Zeichen anzufügen (wordurch die Länge vergrößert wird), bis die Kapazität erreicht ist.

a) Machen Sie sich mit dem Datentyp Str vertraut. Was geschieht, wenn beim Anfügen (append) an einen Str die Kapazität nicht ausreicht? Welche Einschränkung hat der gewählte Ansatz? Schreiben Sie die Antwort als Kommentar in die C-Datei.

b) Implementieren Sie die Funktion **split_words**. Diese bestimmt die in einem Text vorkommenden Wörter als Array von Strs. Die Eingabe ist unformatierter Text, bei dem Wörter durch Leerzeichen und Zeilenumbrüche (siehe **skip_word**) voneinander getrennt sind. Der Ausgabeparameter word muss auf ein Array der Länge `words_length` zeigen. Die Elemente des Arrays sollen von **split_words** entsprechend gesetzt werden. Die Funktion soll die Anzahl der gesetzten Elemente zurückgeben. Die Funktion muss abbrechen, wenn die Arraylänge `words_length` erreicht ist.

   Tipp: Die Implementierung kann ähnlich zur Implementierung von **count_words** realisiert werden.

c) Implementieren Sie die Funktion **words_align_left**. Diese erhält ein Array mit Wärtern als Eingabe und soll daraus einen formatierten Text der Breite width generieren und in dst (destination) ablegen. Die Kapazität von dst muss genügend groß sein.

Aufgabe 3: Binärbäume in Dateien speichern

In dieser Aufgabe geht es darum, Binärbäume, die Integer-Werte enthalten, in Dateien zu speichern und wieder aus Dateien auszulesen. Die Bäume sollen als Binärdaten in den Dateien gespeichert werden. Das Dateiformat soll sein: Ein leerer Baum bzw. Knoten (NULL) wird durch ein 0-Byte repräsentiert. Ein nichtleerer Knoten wird als ein 1-Byte gefolgt von dem Integer-Wert des Knotens, gefolgt vom linken Unterbaum, gefolgt vom rechten Unterbaum repräsentiert.

Hier einige Beispiele für Binärbäume und deren Repräsentation in der Datei. Die Repräsentation in der Datei ist dabei als Hexdezimalwert angegeben:

```
- leerer Baum (NULL) → 00 (Dateilänge: 1 byte)

- Baum, der nur aus der Wurzel mit Wert 0x12345678 besteht:
    (_, 0x12345678, _) →
    01 12 34 56 78 00 00 (Dateilänge: 7 byte)

- Baum mit dem Wert 0x12345678 in der Wurzel und einem linken Unterbaum, der nur
    aus einem Knoten mit dem Wert 0x22222222 besteht:
    ((_, 0x22222222, _), 0x12345678, _) →
    01 12 34 56 78
       01 22 22 22 22 00 00
       00 (Dateilänge: 13 byte)

- Baum mit dem Wert 0x33333333 in der Wurzel und einem rechten Unterbaum, der
   nur aus einem Knoten mit dem Wert 0x44444444 besteht:
    Baum (_, 0x33333333, (_, 0x44444444, _)) →
    01 33 33 33 33
       00
       01 44 44 44 44 00 00 (Dateilänge: 13 byte)

- Baum mit dem Wert 0x12345678 in der Wurzel und nicht-leeren Unterbäumen:
    (((_, 0x44444444, _), 0x22222222, _), 0x12345678, (_, 0x33333333, (_, 0
   x55555555, _))) →
    01 12 34 56 78c
       01 22 22 22 22
          01 44 44 44 44 00 00
          00
       01 33 33 33 33
          00
          01 55 55 55 55 00 00 (Dateilänge: 31 byte)
```

a) Implementieren Sie die Funktion **write_tree_rec**. Verwenden Sie die Funtionen **write_byte** und **write_int** zum Schreiben von Bytes bzw. Integer-Werten in die Datei. Da bei Ein- und Ausgabeoperationen Fehler auftreten können, muss geprüft werden, ob die Operation erfolgreich war und im Fehlerfall mit der Rückgabe von false abgebrochen werden.

b) Implementieren Sie die Funktion **read_tree_rec**. Verwenden Sie die Funtionen **read_byte** und **read_int** zum Lesen von Bytes bzw. Integer-Werten aus der Datei. Da bei Ein- und Ausgabeoperationen Fehler auftreten können, muss geprüft werden, ob die Operation erfolgreich war und im Fehlerfall mit der Rückgabe von false abgebrochen werden.

Ignorieren Sie die Tatsache, dass die Reihenfolge der Bytes eines Integer-Wertes in der Datei davon abhängt, ob das Programm auf einer **little-endian** oder **big-endian-Architektur** ausgeführt wird. Bei little-endian würde der Integer-Wert **0x12345678** gespeichert werden als **0x12 0x34 0x56 0x78**, bei big-endian **0x78 0x56 0x34 0x12**.

Hinweis: `FILE*` funktioniert in C ähnlich zum In- und Output zur/von der Kommandozeile. Das heißt, Sie müssen den Wert `file` an keiner Stelle verändern, sondern können diesen immer unverändert (auch bei rekursiven Aufrufen) übergeben.