Dr. habil. Bernd Schürmann
Dr. Annette Bieniusa
pinc-support@cs.uni-kl.de

TU Kaiserslautern
Fachbereich Informatik

Übungsblatt 10: Programmieren in C (WS 2019/20)

Abgabe: Montag, 27.01.20, 12:00

Aufgabe 1 *Zufallszahlen*

Abgabe: `zufall.c` **(4 Punkte)**

Die Bibliothek `stdlib.h` enthält einen Pseudo-Zufallszahlengenerator, mit Hilfe dessen man zufällig Integer zwischen 0 und `RAND_MAX` (Konstante, definiert in `stdlib.h`) erzeugen kann.

Der Pseudo-Zufallszahlengenerator muss zunächst (einmal!) über `srand(unsigned integer seed)` initialisiert. Verwendet man für verschiedene Programmaufrufe den gleichen Wert für den Parameter `seed`, wird dann die gleiche Folge von Zufallszahlen erzeugt.[1]

Durch jeden folgenden Aufruf von `int rand (void)` wird dann eine neue Pseudo-Zufallszahl erzeugt.

1. Schreiben Sie eine Funktion `int rand_interval(int von, int bis)`, die eine Zufallszahl in einem gegebenen Intervall `[von,bis]` erzeugt.
2. Für ein Würfelspiel wollen Sie einen Würfel simulieren und dazu Zahlen zwischen 0 und 9 zufällig erzeugen. Damit das Spiel fair ist, sollten alle Zahlen mit gleicher Wahrscheinlichkeit erzeugt werden. Um zu testen, wie gut der Generator ist, simulieren Sie 10.000.000 Würfelereignisse und zählen, wie häufig die einzelnen Zahlen gewürfelt werden.

   Für die Abgabe in Exclaim verwenden Sie bitte als Seed für den Generator den Wert `123456` und erzeugen eine Ausgabe der folgenden Art:

```
0 : 999896
1 : 999745
2 : 1000466
3 : 1002458
4 : 1000228
5 : 999964
6 : 1000173
7 : 1001123
8 : 997263
9 : 998684
```

Aufgabe 2 *Bitmanipulation*

Abgabe: `schluessel.c` **(8 Punkte)**

Ein digitales Schloss habe 32 Stellen mit den Werten 0 oder 1, z.B.

```
Schloss = 1 0 0 1 0 1 1 1 1 0 0 1 0 1 0 1 0 0 0 1 1 0 1 1 1 1 1 1 0 0 0 0
```

Ein digitaler Schlüssel `schluessel` öffnet das Schloss, falls der Schlüssel eine 1 an den Stellen hat, an denen das Schloss eine 0 hat, und umgekehrt. Der Schlüssel

```
0 1 1 0 1 0 0 0 0 1 1 0 1 0 1 0 1 1 1 0 0 1 0 0 0 0 0 0 1 1 1 1
```

[1] Die Verwendung von gleichen Zufallssequenzen ist hilfreich, um reproduzierbare Tests zu schreiben. Will man tatsächlich zufällige Sequenzen, kann beispielsweise die aktuelle Zeit als Initialisierungswert verwenden werden: `srand(time(NULL))` verwendet die Anzahl der Sekunden seit Januar 1, 1970 als Initialisierungswert.

würde oben genanntes Schloss öffnen. Zwei Integer-Variablen sollen das Schloss und den Schlüssel implementieren:

```
uint32_t schloss, schluessel;
```

In dieser Aufgabe soll der Schlüssel zu einem gegebenen Schloss gefunden werden, wobei der Code des Schlosses nicht bekannt ist. Hierzu soll es eine Funktion

```
int count_correct_positions (uint32_t ts);
```

die einen Testschlüssel `ts` als Eingabe erhält und als Ergebnis die Anzahl korrekter Stellen zurückgibt. In der main-Funktion soll damit der für ein gegebenes Schloss passende Schlüssel gefunden werden.

1. Verständnisfragen (ohne Abgabe):
   - Falls das Schloss folgenden Wert zugewiesen bekommt, welchen Wert hat dann der Schlüssel?

     ```
     schloss = 0xA51F4700;

     schluessel = 0x________________________
     ```

   - Warum ist es sinnvoller die Variablen `schloss` und `schluessel` als unsigned int Datentyp zu implementieren und nicht als `int`-Werte bzw. Arrays mit boolschen Werten?
2. Schreiben Sie die oben genannte C-Funktion `count_correct_positions`.

   ```
   int count_correct_positions(uint32_t ts) {
       // Anlegen einer Integer-Variable schloss als globale Variable,
       // die während der gesamten Programmlaufzeit existiert,
       // aber nur innerhalb dieser Funktion sichtbar ist.
       // Diese wird zu Beginn einmal mit einer Zufallszahl initialisiert.
       static uint32_t schloss = 0;
       if (!schloss)
       {
           schloss = rand();
       }

       // Bitte hier ergaenzen!
   }
   ```

3. Schreiben Sie die `main`-Funktion zur Berechnung des Schlüssels unter Verwendung der Funktion `count_correct_positions`. Beschreiben Sie zunächst einen effizienten Algorithmus zu Berechnung des Schlüssels in Worten (als Kommentar im Code). Implementieren Sie dann die Funktion. Der Zufallszahlengenerator soll dabei mit `111111` initialisiert werden. Die Ausgabe des Programmes soll folgendermaßen sein:

   ```
   Der gesuchte Schluessel ist 0x...
   ```

   Dabei soll `0x...` durch den Schlüssel im Hexadezimal-Format ersetzt werden.

   Hinweis: Zum Testen können Sie das Schloss zunächst mit `0xffffffff` initialisieren.

Aufgabe 3 *Datenkompression*

Abgabe: `compress.c` (10 Punkte)

Daten können häufig in weniger Speicherplatz abgespeichert werden, wenn sie aus einem kleineren Wertebereich stammen. In dieser Aufgabe entwickeln wir ein Programm, das Texte in ASCII komprimiert darstellt. Zur Vereinfachung nehmen wir an, dass der Text in einem 7-Bit ASCII Format codiert ist.

Der Komprimierungsalgorithmus geht folgendermaßen vor:

- Wenn mehrere Kleinbuchstaben hintereinander vorkommen, wird jedes dieser Zeichen aus der 7-Bit-Darstellung als `char` in eine 5-Bit-Darstellung umgewandelt und jeweils 3 Zeichen in einem 16-Bit Wert (`uint_16`) zusammengefasst. Ein solcher Dreier-Block wird dabei binär mit der Maske `0x8000` gekennzeichnet; d.h. das führende Bit ist 1.
- Alle anderen Zeichen werden in der ursprünglichen Codierung dargestellt und dabei immer zu Zweier-Blöcken in ein 16-Bit-Wert zusammengefasst.

Beispiele:

- Das Bitmuster `1000 0100 0100 0011` codiert die Zeichen `bcd`.
- Das Bitmuster `0110 0001 0011 0101` codiert die Zeichen `a5`.

Implementieren Sie die beiden folgenden Funktionen:

```
int decompress(uint16_t *input, int input_size, char *output);
int compress(char *input, uint16_t *output);
```

Die Funktionen sollen jeweils die Eingabe `input` nehmen und komprimieren bzw. dekomprimieren. Das Ergebnis soll in `output` abgespeichert werden. Rückgabewert ist die Größe des Ergebnisses (d.h. wie weit der `output` gefüllt ist). Falls eine Umwandlung nicht möglich ist, soll das Programm die folgende Fehlermeldung `Umwandlung nicht möglich` ausgeben und dann abbrechen.

Auf der Vorlesungsseite finden Sie eine Vorlage.

Hinweis: Um Ihre Lösung lokal zu testen, sollten Sie Ihren Eingabestring in einer Textdatei (z.B. `eingabe.txt` abspeichern und diese bei der Ausführung auf Kommandozeile folgendermaßen als Eingabe nutzen:

```
 > gcc -o compress compress.c
> ./compress < eingabe.txt
```

Erklärung: Um einen String auf sichere Art einzulesen, verwenden wir die Funktion `fgets(char * str, int n, FILE * stream)`, die `n` Zeichen aus einem Eingabestrom in den Speicherbereich schreibt, der durch `str` referenziert wird. Diese Funktion nimmt einen Buffer `s` der Größe `n` und einen Eingabe-Strom. Wenn Sie `stdin` als Eingabe-Strom verwenden, wird von der Standardeingabe gelesen. Die Funktion liest bis eine neue Zeile startet, das Ende des Stroms erreicht ist oder die maximale Länge `n-1` erreicht ist. Rückgabe der Funktion ist `s`, falls das Einlesen erfolgreich war und `NULL` falls das Ende des Stroms erreicht wurde oder ein Fehler auftritt. Zeilenumbrüche am Ende der Zeile werden in `s` beibehalten.