Heinrich Heine Universität Düsseldorf

Hardwarenahe Programmierung - Übungsblatt 1

- Es wird vier Übungsblätter mit jeweils 3 Pflichtaufgaben und ein Abschlussprojekt geben. Um die Veranstaltung zu bestehen müssen Sie:
  - mindestens 9 von insgesamt 12 Pflichtaufgaben **fehlerfrei** lösen,
  - auf **jedem** Übungsblatt mindestens eine Aufgabe **fehlerfrei** lösen und
  - das Abschlussprojekt bestehen.
- Im Gegensatz zu Pflichtaufgaben werden Vorbereitungsaufgaben nicht korrigiert. Um die Pflichtaufgaben lösen zu können ist es trotzdem ratsam, sich vorher mit den Vorbereitungsaufgaben zu befassen.
- Hinweise:
  - Beginnen Sie frühzeitig mit den Übungsaufgaben! Wenn Sie zu spät anfangen haben Sie nicht mehr die Möglichkeit, Fragen zu stellen.
  - Wenn ein Programm die Tests besteht bedeutet das noch nicht, dass es auch wirklich fehlerfrei ist. Achten Sie besonders auf Speicherzugriffsfehler!
  - Wir werden zusätzliche (geheime) Testfälle prüfen.
  - Nutzen Sie die Compiler-Optionen `-g -fsanitize=address -fsanitize=undefined` und das Programm `valgrind`, um leichter Speicherzugriffsfehler zu finden.
  - Die Verwendung von nicht-initialisierten Variablen und Speicherzugriffsfehler aller Art führen zum Nichtbestehen der Aufgabe, auch wenn die Tests zufällig durchlaufen sollten.
  - Abschreiben und abschreiben lassen ist verboten. Selbst einzelne Zeilen dürfen nicht aus dem Internet oder aus Büchern kopiert werden, auch nicht mit Quellenangabe.
  - Aufgaben müssen **selbstständig** bearbeitet werden. Die Aufgaben sollen *Ihre* Leistung beurteilen und nicht die Ihrer Mitmenschen.
  - Zippen Sie zur Abgabe den gesamten Ordner und laden Sie ihn im ILIAS hoch. Stellen Sie vorher sicher, dass keine Dateien doppelt abgegeben werden, auch nicht in verschachtelten Zip-Archiven. Bei doppelt vorkommenden Dateinamen wir eine zufällige Datei ausgewählt und korrigiert.
  - Laden Sie Ihre Abgabe im ILIAS zur Sicherheit nochmal runter, um zu überprüfen, ob Sie wirklich die richtige Abgabe hochgeladen haben und nicht etwa veraltete Fassungen oder ungelöste Aufgabenstellungen.

  Termine für Einzelgespräche:
  - Mittwoch, der 13. Oktober, von 12:30 Uhr bis 16:00 Uhr.
  - Donnerstag, der 14. Oktober, von 10:30 Uhr bis 14:00 Uhr.
  - Freitag, der 15. Oktober, von 8:30 Uhr bis 12:00 Uhr.

Fragen und Probleme

Wenn Sie bei einer Aufgabe nicht weiter kommen haben Sie die Möglichkeit, ihre Probleme in einem Einzelgespräch zu erörtern.

Zu Beginn der Gesprächtszeiten wird im RocketChat ein Beitrag eröffnet. Antworten Sie mit einem beliebigen Text oder Emoji auf diesen Beitrag, wenn Sie Hilfe benötigen. Hierüber wird die Reihenfolge verwaltet. Im BigBlueButton-Raum werden sie dann zu einem Einzelgespräch eingeladen, wenn Sie dran sind.

- Außerhalb der Zeiten werden keine Fragen beantwortet. Beschäftigen Sie sich deshalb frühzeitig mit den Aufgaben und nicht erst am Wochenende.
- Die TutorInnen werden die Aufgaben nicht für Sie programmieren.
- Einzelgespräche sind beschränkt auf höchstens 15 Minuten. Fassen Sie deshalb Ihre Probleme so klar wie möglich zusammen, dass wir diese schnell lösen können:
  - Was haben Sie probiert?
  - Welches Resultat haben Sie erwartet?
  - Was passiert stattdessen? Haben Sie ihren Code Schritt für Schritt nachverfolgt? Geben Sie alle Zwischenergebnisse aus, bis Sie genau wissen, bei welchem Schritt ihr erwartetes Resultat vom tatsächlichen Resultat abweicht.
  - Wird beim Kompilieren und Ausführen mit den folgenden `gcc`-Optionen eine Zeilenzahl ausgegeben, in der etwas schief geht?

    ```
    -g -fsanitize=address -fsanitize=undefined -Wall -Wextra -std=c99
    ```
  - Gibt es Fehlermeldungen beim Ausführen des Programms mit `valgrind`?
  - Haben Sie sich vorher über das Thema informiert? Haben Sie die Dokumentation der Funktionen gelesen, die Sie verwenden wollen? Haben Sie die Abschnitte zu den entsprechenden Themen in den Büchern gelesen? Haben Sie nach der Fehlermeldung im Internet gesucht?
- Zusätzliche Einzelgespräche sind beschränkt auf 5 Minuten, falls an dem Tag noch Zeit ist.
- Reduzieren Sie ihre Probleme so weit wie möglich. Erstellen Sie ein Minimal, Reproducible Example. Im besten Fall haben Sie eine einzelne Datei mit unter 15 Zeilen, mit der Sie Ihr Problem schildern können.
- "Die Tests schlagen irgendwo fehl." ist keine ausreichende Fehlerbeschreibung. Isolieren Sie den genauen Testfall in einer neuen Datei.
- "`gcc`/`valgrind`/`fsanitize` meckert." ist keine ausreichende Fehlerbeschreibung. Wie lautet die genaue Fehlermeldung?

Vorbereitungsaufgabe 1 Grundlegende Datentypen

In C gibt es eine Vielzahl an Datentypen, zum Beispiel für vorzeichenlose ("*unsigned*") Integer, vorzeichenbehaftete Integer und Gleitkommazahlen, die alle für unterschiedliche Zahlenbereiche geeignet sind.

In der Datei `datatypes/datatypes.c` können Sie sich dazu ein paar Beispiele ansehen.

Vorbereitungsaufgabe 2 Speicher und Zeiger

In dieser Aufgabe werden einige Beispiele für die Speicherbereiche "Stack" und "Heap" in C sowie häufige Fehler vorgestellt. Lesen Sie sich die gegebenen C-Dateien im Verzeichnis `memory` durch, damit Sie mit den häufigsten Fehlern in C vertraut sind.

1. In `pointer_examples.c` finden Sie Beispiele zu Zeigern.
2. In `pointer_parameter.c` finden Sie Beispiele für die Verwendung von Zeigern als Funktionsparameter.
3. In `pointer_return_value.c` finden Sie Beispiele für Zeiger als Rückgabewerte.
4. In `pointer_scanf.c` finden Sie Beispiele für das Einlesen von Integern mit der Funktion `scanf`.
5. In `stackoverflow.c` finden Sie ein Beispiel, wie man einen Stackoverflow erzeugt.

- Kompilieren Sie die Programme mit dem folgenden Befehl und führen Sie sie aus. Schlagen diese bei Fehlern sofort fehl? Ersetzen Sie `datei` durch den entsprechenden Programmnamen.

  ```
  gcc -Wall -Wextra -std=c99 -g programm.c datei.c -o programm
  ```

- Führen Sie die Programme nun mit `valgrind ./programm`. Gibt es jetzt mehr Fehlermeldungen?
- Kompilieren Sie die Programme nun mit dem folgenden Befehl[1] und führen Sie sie aus. Gibt es hierbei andere Fehlermeldungen?

  ```
  gcc -g -fsanitize=address -fsanitize=undefined datei.c -o datei
  ```

Hinweis: Es gibt Fehler, die nicht mit `valgrind` und auch nicht mit `fsanitize` erkannt werden. Diese Programme *unterstützen* Sie deshalb nur bei der Suche nach Speicherzugriffsfehlern, aber nehmen Ihnen nicht die ganze Arbeit ab.

Vorbereitungsaufgabe 3 Unit-Tests

Um Fehler bei der Entwicklung eines Programms zu vermeiden ist es wichtig, dass man die Ausgabe von Funktionen einfach und schnell überprüfen kann. Hierzu sind Unit-Tests hilfreich.

In dieser Veranstaltung verwenden wir hierzu das "Check unit testing framework".

1. Tests werden dabei in einer `ts`-Datei geschrieben und mit dem Programm `checkmk` zu `c`-Dateien konvertiert. Am Beispiel der Datei `add_tests.ts`, die wir Ihnen im Verzeichnis `unittests` zur Verfügung stellen, sieht das so aus:

   ```
   checkmk add_tests.ts > add_tests.c
   ```

   Das Programm `checkmk` liest den Inhalt der Datei `add_tests.ts` ein und gibt den generierten C-Code aus. Diese Ausgabe wird mit dem Operator "`>`" umgeleitet in die Datei `add_tests.c`.
2. Weiterhin wird eine Datei mit der Implementierung der zu testenden Funktion benötigt. In unserem Beispiel heißt die Datei `add.c` und soll zwei Zahlen addieren.
3. Zudem gibt es noch die Datei `add.h`, in der die Funktion deklariert wird. Funktionen müssen in C deklariert werden, bevor sie verwendet werden können. Die tatsächliche Implementierung (der Funktionskörper) darf später folgen. Die Datei `add_tests.ts` inkludiert die Datei `add.h`, damit die Funktion bekannt ist und verwendet werden kann.
4. Um das Check-Framework zu verwenden müssen dem Compiler `gcc` einige Parameter übergeben werden. Diese variieren von System zu System, können aber automatisch mit dem Befehl `pkg-config -libs -cflags check` generiert werden. Um die Ausgabe dieses Programms nicht jedes Mal neu abtippen zu müssen speichern wir sie in der Variable `BUILDFLAGS`.

[1] Der `AddressSanitzer`, der mit `fsanitize` aktiviert wird, ist leider inkompatibel mit `valgrind`. Wenn Sie sowohl den `AddressSanitizer` als auch `valgrind` nutzen möchten müssen Sie also zwei mal kompilieren.

```
BUILDFLAGS=$(pkg-config --libs --cflags check)
```

5. Bei Bedarf können wir uns den Wert dieser Variablen ausgeben lassen.

```
echo $BUILDFLAGS
```

6. Der Compiler `gcc` kann vor häufigen Fehlern warnen und einige Speicherzugriffsfehler erkennen, wenn ihm entsprechende Anweisungen dazu übergeben werden. Auch die werden wir in einer Variable speichern.

```
CFLAGS="-Wall -Wextra -Werror -std=c99 -g -fsanitize=address -fsanitize=undefined"
```

7. Nun kann die Funktionsimplementierung zusammen mit den Tests kompiliert werden. Hierbei wird das Programm `tests` erzeugt. Der Inhalt der Variablen `CFLAGS` und `BUILDFLAGS` wird mit dem Operator `$` automatisch in den Aufruf des Programms eingefügt.

```
gcc $CFLAGS add.c add_tests.c -o tests $BUILDFLAGS
```

8. Nach der Kompilierung können die Tests mit dem Befehl `./tests` ausgeführt werden. In der von uns bereitgestellten Implementierung in der Datei `unittests/add.c` gibt es einen Fehler, weshalb einer der Tests fehlschlägt. Finden Sie den Fehler?

9. Manchmal hilft es bei der Fehlersuche, wenn man eine Funktion unabhängig von den Tests kompilieren und ausprobieren kann. Dazu haben wir die Datei `unittests/main.c` bereitgestellt. Hiermit können wir ein weiteres Programm `main` erstellen und ausführen. Spätestens danach sollte der Fehler offensichtlich sein.

```
gcc $CFLAGS add.c main.c -o main
./main
```

Pflichtaufgabe 4 Refactoring

In Programmen ist es allgemein sehr sinnvoll Funktionen zu schreiben, um eine gute Strukturierung und Lesbarkeit vorauszusetzen. Sie bekommen von uns ein funktionsfähiges Programm mit nur einer `main`-Funktion, welche allerdings sehr unübersichtlich ist. Das Programm finden Sie in den Dateien `refactoring/laden.c` und `refactoring/laden.h`.

Ihre Aufgabe ist es, dieses Programms in mehrere Unterfunktionen aufzuteilen, um so das von uns gesetzte Zeilenlimit pro Funktion durchsetzen zu können.

- Das gesetzte Zeilenlimit pro Funktion beträgt in dieser Aufgabe 27 Zeilen, abweichend von den gewöhnlichen 60 Zeilen in den Code-Vorgaben.
- Alle von Ihnen erstellten Funktionen müssen in der Datei `laden.h` *deklariert* und diese in der C-Datei *inkludiert* werden.
- Benutzen Sie in Ihren Funktionen Zeiger und Pointer!

Der Sinn dieser Aufgabe ist es nicht, die main Funktion zu reduzieren ohne Unterfunktionen zu schreiben. Sie dürfen den Code aber überarbeiten und gegebenenfalls verbessern.

Testen Sie Ihr Programm mit dem Befehl `./test.sh`

Pflichtaufgabe 5 Maximum von Teilarrays mit Pointerarithmetik

In C müssen Sie die Länge eines Arrays immer mitführen und an entsprechende Funktionen übergebenen. Während dies zwar zusätzlichen Programmieraufwand und eine zusätzliche Fehlerquelle bedeutet, entsteht dadurch zusammen mit der Pointerarithmetik auch ein gewisses Maß an Flexibilität.
In dieser Aufgabe sollen Sie Zahlen in ein Array einlesen und anschließend das Maximum des ersten, mittleren und letzten Drittel des Arrays ausgeben.

```
Beispiel-Array mit 15 Elementen
      |----max: 7----|----max: 9----|------max: 8------|
Array: 5, 6, 7, 1, 2, 3, 4, 8, 9, 4,  2,  8,  3,  4,  1
Index: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14
      |---erste 5----|--mittlere 5--|-----letzte 5-----|
```

Arbeiten Sie in dieser Aufgabe **ausschließlich** mit Pointerarithmetik und verzichten Sie komplett auf die Arrayschreibweise mit eckigen Klammern `[]`.

1. Schreiben Sie in der Datei `arrays/arrays.c` zunächst eine Funktion

   ```
   void read_numbers(int *array, size_t length);
   ```

   welche die entsprechende Anzahl an Zahlen in das Array einliest. Sie können davon ausgehen, dass nur gültige Zahlen in Ihr Programm eingegeben werden [2].

2. Anschließend schreiben Sie die Funktion

   ```
   int max(int *array, size_t length);
   ```

   welche das Maximum aus dem gegebenen Array findet und dieses zurückgiebt. Es wird immer ein Array mit mindestens einem Element übergeben.

3. Als letztes implementieren Sie die Funktion

   ```
   void print_max_of_thirds(int *array, size_t length);
   ```

   Diese soll die Maxima der drei Teilarrays nacheinander auf der Konsole ausgeben. Jedes Teilarray ist dabei genau ein Drittel des ursprünglichen Arrays. Sie dürfen davon ausgehen, dass `length` immer durch 3 teilbar und immer mindestens 3 ist.
   Die Funktion **muss** Ihre vorher implementierte `max` Funktion verwenden um die Maxima zu bestimmen!
   Die Ausgabe soll folgendes Format haben:

   ```
   Maximum 1. Drittel: 7
   Maximum 2. Drittel: 9
   Maximum 3. Drittel: 8
   ```

4. Testen Sie Ihre Abgabe indem Sie das Testskript mit dem Befehl `./test.sh` ausführen.
   Sie können und sollen Ihr Programm aber auch zwischendurch schon selbstständig testen und debuggen. Dazu können Sie beispielsweise eine weitere C-Datei erstellen und dort eine eigene `main`-Funktion zu Testzwecken schreiben.

[2] Der Datentyp `size_t` ist ein vorzeichenloser (*unsigned*) Integer. Der C99-Standard garantiert, dass der Datentyp groß genug ist, um jeden gültigen Array-Index speichern zu können. Ein `int` wäre hierfür nicht unbedingt geeignet. Zum Beispiel kann ein `int` auf der VM höchstens den Wert 2147483647 speichern. Größere Arrays könnten mit der Funktion also nicht verarbeitet werden, wenn für die Länge der Datentyp `int` verwendet worden wäre.

Pflichtaufgabe 6 Arrays umdrehen

In dieser Aufgabe geht es um das Umdrehen von (Teil-)Arrays. Hierbei ist ein Index in einem Array gegeben, ab dem eine bestimmte Anzahl an Array-Werten umgedreht werden soll.

In folgendem Beispiel werden 4 Werte beginnend bei Index 3 umgedreht. Im betrachteten Teil-Array $1, 2, 3, 4$ wird der erste Wert 1 mit dem letzten Wert 4 und der zweite Wert 2 mit dem vorletzten Wert 3 vertauscht. Das umgedrehte Teil-Array enthält also danach die Zahlenfolge $4, 3, 2, 1$.

```
                    Diese 4 Werte werden umgedreht.
                    |  |  |  |
Array    : 5, 6, 7, 1, 2, 3, 4, 8, 9
Umgedreht: 5, 6, 7, 4, 3, 2, 1, 8, 9
Index    : 0, 1, 2, 3, 4, 5, 6, 7, 8
                    |
                    Beginnend bei Index 3.
```

Hierzu soll die folgende Funktion verwendet werden:

```
void reverse(int array[], size_t length, size_t index, size_t count);
```

Die Funktion nimmt ein Array, die Länge dieses Arrays, einen Startindex und die Anzahl der umgedrehten Werte entgegen. Ein Funktionsaufruf könnte für das vorherige Beispiel so aussehen:

```
reverse(array, 9, 3, 4);
```

In den Dateien `reverse_recursive.c`, `reverse_recursive.h`, `swap.c` und `swap.h` ist Ihnen eine Implementierung für eine solche Funktion gegeben.

1. In der Datei `reverse_tests.ts` befinden sich Unit-Tests, aus denen mit dem Programm `checkmk` eine C-Datei erzeugt werden kann. Wenn Sie diese C-Datei zusammen mit der rekursiven Implementierung kompilieren und ausführen, dann sollten diese Tests fehlerfrei durchlaufen.

   Leider sind die Tests unvollständig und decken einige wichtige Fälle nicht ab. Ergänzen Sie die Datei `reverse_tests.ts` um mindestens einen weiteren Test, der einen Fehler in der Implementierung aufdeckt. Die Tests sollten nun fehlschlagen.

2. Korrigieren Sie den Fehler in der Implementierung. Die ergänzten Tests sollten nun fehlerfrei durchlaufen. Sie dürfen die Funktion `reverse` nicht komplett neu implementieren, sondern nur den Fehler korrigieren.

3. Erzeugen und kompilieren Sie die rekursive Implementierung zusammen mit den Tests in der Datei `reverse_long_tests.ts`. Bei diesen Tests wird ein Array mit 10 Millionen Werten umgedreht. Leider schlagen diese Tests fehl. Wenn Sie die Ursache dafür noch nicht verstehen betrachten Sie nochmals die Vorbereitungsaufgabe “Speicher und Zeiger”.

4. Implementieren Sie in der Datei `reverse_iterative.c` eine *iterative* Funktion zum Umdrehen von Arrays. Diese Funktion sollte die Tests in beiden Test-Dateien bestehen.

   Die Funktion darf auch bei falscher Länge, Index oder Anzahl keinen Speicherzugriffsfehler verursachen. Wenn ein Index die gegebene Länge überschreiten würde soll die Funktion keinen einzigen Wert vertauschen und ansonsten nichts tun. Es darf angenommen werden, dass `array` bei ausreichender Länge, Index und Anzahl auf einen gültigen Speicherbereich verweist.

5. Prüfen Sie die Code-Vorgaben, indem Sie den Befehl `./codestyle.py` ausführen.