Institut für Informatik
Prof. Dr. Stefan Conrad

Sommersemester 2020
Übung 4

Hardwarenahe Programmierung
Gruppe 14 (Lars)

In dieser Übung nehmen Sie eine high-level-Perspektive ein und beschäftigen sich mit dem C-Kompilierprozess und dessen technischen Grundlagen wie der Repräsentation von unterschiedlichen Datentypen. Damit zusammenhängend arbeiten Sie weiter mit `make` *und dem C-Debugger* `gdb`.

Keine Angst – es sind diesmal sehr viele Aufgaben, die meisten davon sind aber sehr schnell zu lösen. Denken Sie wie immer daran, die Lösungen zu den Pflichtaufgaben hochzuladen und den Test im ILIAS zu absolvieren!

Aufgabe 1 *Datentypen und Casts (**Pflichtaufgabe**)*

Betrachten Sie den unten stehenden C-Code.

```
int a = 3;
int b = 2;
double x = a / b;
```

(a) Probieren Sie für sich aus, welchen Wert `x` annimmt.

(b) Implementieren Sie in der Datei `casts.c` die Funktion `dividiere`, die den Quotienten a/b berechnet, ohne dabei zu runden.

```
double dividiere(int a, int b);
```

Aufgabe 2 *Der Compiler (Vorbereitungsaufgabe)*

Recherchieren Sie für Sich die Antworten auf folgende Fragen:

(a) Aus welchen 4 Stufen besteht der C-Kompiliervorgang? Was macht der Compiler in den jeweiligen Stufen?

(b) Sie haben bereits die Compilerdirektive `#include` kennengelernt. Auf welcher der 4 Stufen wird sie verarbeitet? Warum kann diese Direktive nicht in anderen Stufen sinnvoll verarbeitet werden?

Aufgabe 3 *Endianess (Vorbereitungsaufgabe)*

Schreiben Sie ein Programm `endianess`, das die Art der Bytereihenfolge (Endianess) auf Ihrem Computer ermittelt.

Sie können sich bei Bedarf z.B. auf `https://de.wikipedia.org/wiki/Byte-Reihenfolge` über das Konzept von Bytereihenfolge/Endianess informieren.

Beispielaufruf:

```
./endianess
```

Ausgabe: "`0x1234 ist 0x34 0x12 - dieser Computer benutzt little-endian.`"

Aufgabe 4 *Datentypen (Vorbereitungsaufgabe)*

Unter *char.c* kann ein unvollständiges Programm gefunden werden, das Benutzern ausgeben soll, wie viele Bytes der Datentyp `char` zur Speicherung benötigt, und was der kleinste bzw. der größte darstellbare Wert dieses Datentyps ist.

Beispielaufrufe:

`./char`

Ausgabe: "*Der Datentyp char benötigt TODO Bytes Speicherplatz.*"

`./char -m`

Ausgabe: "*Der Datentyp char benötigt TODO Bytes Speicherplatz.*
Die kleinste darstellbare Zahl beträgt dann TODO."

`./char -M`

Ausgabe: "*Der Datentyp char benötigt TODO Bytes Speicherplatz.*
Die größte darstellbare Zahl beträgt dann TODO."

Vervollständigen Sie das Programm. Ersetzen Sie dazu im Programm die Platzhalter "TODO" und "42" und fügen Sie fehlenden Code hinzu.

Aufgabe 5 *Datentypen (Vorbereitungsaufgabe)*

Unter *typen1.c* kann ein unvollständiges Programm gefunden werden, das vom Benutzer eine ganze Zahl einliest und ausgeben soll, welchen Wert eine `char`, `int` oder `double`-Variable annimmt, wenn dieser Wert darin gespeichert wird. Verwandeln Sie den eingegebenen String aus `argv` zuerst in einen `long long`-Wert mithilfe der Funktion `atoll()`.

Beispielaufruf:

```
./fassungsvermoegen 5608973456345
```

Ausgabe:

```
char      : -39
int       : -253832231
double    : 5608973456345.000000
```

Ersetzen Sie dazu im Programm "TODO" und "42" geeignet, und ergänzen Sie ggf. zusätzlich benötigten Code.

Aufgabe 6 *Floss your teeth and make backups! (Vorbereitungsaufgabe)*

Schreiben Sie ein Makefile-Rezept, das eine Sicherheitskopie aller Quell- und Headerdateien aus dem aktuellen Verzeichnis in eine zipdatei packt.

Aufgabe 7 *Make it again, Sam (Vorbereitungsaufgabe)*

Wie kann man make dazu zwingen, beim nächsten Kompilieren alles neu zu kompilieren, und nicht alte Kompilate wiederzuverwenden?

(a) Überlegen Sie sich zunächst, wie Sie dies per Hand (also ohne Makefile) tun würden.

(b) Schreiben Sie nun ein Makefile-Rezept, das diesen Vorgang automatisiert. Nennen Sie Ihr Target `clean`.

Aufgabe 8 *GDB (**Pflichtaufgabe**)*

In dieser Aufgabe sollen Sie den GNU-Debugger `gdb` benutzen, um in einem bereits kompilierten C-Programm (zu dem Ihnen der Quellcode zunächst nicht zur Verfügung steht) den Inhalt einer geheimen Passwort-Variable herauszufinden.

Denken Sie daran, dass Variablen, die nicht initialisiert sind, beliebige Werte annehmen können.

Bei Bedarf finden Sie Links zu hilfreichen Online-Quellen zu `gdb` im Lernmodul.

(a) Starten Sie das von uns vorgegebene Programm `gdb_2` mit folgendem Befehl:

```
gdb gdb_2
```

Hinweis: Bei einigen Linux-Systemen gibt es die Fehlermeldung, dass `gdb_2` nicht gefunden werden kann. In diesem Fall hilft es oft, die 32-bit Version von `libc` zu installieren:

```
sudo dpkg --add-architecture i386
sudo apt-get update
sudo apt-get install libc6:i386
```

Finden Sie nun mit Hilfe von `gdb` heraus, welcher Wert in der Variablen `pwd` steht! Versuchen Sie zuerst die Variablen in Ihren eigenen Programmen auszulesen, um Erfahrung mit `gdb` zu sammeln.

*Hinweis:*Der Fehlerteufel hat sich am Quelltext dieses Programms ausgelassen und an mehreren Stellen einige Zeichen des Code verändert. Deshalb läuft das Programm in eine Endlosschleife– das sollte Sie nicht aufhalten! Wenn Sie das Programm allerdings nicht starten können, dann liegt dies möglicherweise daran, dass Sie nicht das vorgegebene Betriebssystem verwenden.

(b) Nutzen Sie den Wert von `pwd` als Passwort, um die verschlüsselte Datei `gdb_2.c.gpg` mittels `gpg` zu entschlüsseln, um so an den C-Quellcode des Programms zu gelangen. Sie können dies mit folgendem Befehl tun:

```
gpg -d gdb_2.c.gpg > gdb_2_loesung.c
```

(c) Jetzt, da Sie den Quellcode des Programms haben, ändern Sie den Code so, dass die erwartete Ausgabe produziert wird. Hierbei dürfen nicht mehr als 20 Zeichen verändert werden. Überprüfen Sie mit dem Script `test.sh`, ob Sie die richtige Ausgabe erzeugen.