Institut für Informatik
Prof. Dr. Stefan Conrad

Sommersemester 2021
Übung 4

**Hardwarenahe Programmierung**
Gruppe 11 (Thomas)

*In dieser Übung beschäftigen Sie sich mit dem Kompilieren mehrerer Dateien mit* `make` *und dem C-Debugger* `gdb`

*Denken Sie wie immer daran, die Lösungen zu den Pflichtaufgaben hochzuladen und den Test im ILIAS zu absolvieren!*

## Aufgabe 1 *Kompilieren mehrerer Dateien (**Pflichtaufgabe**)*

Es ist oft sinnvoll, ein Programm auf mehrere C-Dateien aufzuteilen, zum Beispiel wenn dieses sehr lang wird. Um Funktionen in anderen C-Dateien verwenden zu können, sind Funktionsdeklarationen notwendig. Diese werden in der Regel in Header-Dateien aufgeführt.

In dieser Aufgabe erhalten Sie fünf C-Dateien. In diesen Dateien sind Funktionen definiert, die weitere Funktionen aus anderen C-Dateien aufrufen.

- `mathematik.c` beinhaltet die Funktionen `wurzel` und `quadrat`.
- `berechnung1.c` beinhaltet die Funktion `berechnung1`, die drei weiteren Funktionen `quadrat`, `wurzel` und `berechnung3` aufruft.
- `berechnung2.c` beinhaltet die Funktion `berechnung2`, die `wurzel` und `berechnung1` aufruft.
- `berechnung3.c` beinhaltet die Funktion `berechnung3`, die `berechnung2` aufruft.
- `berechnung.c` beinhaltet die main-Funktion, die `berechnung1` und `berechnung2` aufruft.

Leider sind die dazugehörigen Header-Dateien verloren gegangen.

(a) Ergänzen Sie die fehlenden Header-Dateien `mathematik.h`, `berechnung1.h`, `berechnung2.h` und `berechnung3.h`.

(b) Geben Sie einen einzeiligen gcc-Befehl an, der die Dateien zum Programm `berechnung` kompiliert und schreiben Sie den Befehl in die Datei `kompilieren_einzeiler.sh`.

(c) Als Vorbereitung für die nächste Aufgabe ist es notwendig, Dateien einzeln zu Objektdateien zu kompilieren und danach zu linken. Kompilieren Sie die fünf gegebenen C-Dateien einzeln zu Objekt-Dateien und linken Sie diese abschließend zum Programm `berechnung` zusammen. Schreiben Sie die die Befehle in die Datei `kompilieren_einzeln.sh`.

(d) Testen Sie Ihre Lösung mit dem Skript `test.sh`.

**Aufgabe 2** *Make (**Pflichtaufgabe**)*

Um das Kompilieren von mehreren Dateien bequemer zu handhaben, gibt es das Programm `make`. Es ist besonders nützlich, wenn man nur Dateien neu kompilieren möchte, die sich geändert haben.

In dieser Aufgabe wird dies am Beispiel von einem Zoo geübt.

- Erstellen Sie eine Makefile-Datei mit Regeln, die die C-Dateien `elefant.c`, `giraffe.c`, `zoo.c` und `main.c` einzeln zu Objekt-Dateien kompilieren. Weiterhin sollen die Dateien neu kompiliert werden, wenn sich eine Abhängigkeit ändert. Erstellen Sie hierzu passende Regeln. Die erzeugten Objekt-Dateien sollen die gleichen Namen wie die zugrundelegenden C-Dateien haben.

  Zum Beispiel würde die Datei `zoo.c` neu zur Objekt-Datei `zoo.o` kompiliert werden, wenn sich die Abhängigkeiten `zoo.c`, `zoo.h`, `elefant.h` oder `giraffe.h` ändern.
- Weiterhin soll es eine Regel `main` geben, die aus den Objekt-Dateien das Programm `main` erzeugt.
- Es soll eine Regel `all` geben, die von der Regel `main` abhängt.
- Es soll eine Regel `clean` geben, die alle erzeugten Dateien entfernt.
- Das Skript `test.sh` soll zum Testen verwendet werden.

**Anmerkung**: `gcc` kann Abhängigkeiten dynamisch bestimmen („*Auto-Dependency Generation*“). Dies ist ein sehr fortgeschrittenes Thema. Bei dieser Aufgabe reicht es, wenn Sie feste Dateipfade angeben.

**Aufgabe 3** *GDB (**Pflichtaufgabe**)*

Sie erhalten die Datei `roulette.c`. Das daraus resultierende Programm soll nacheinander mehrere Roulette-Zahlen zwischen 0 und 36 (einschließlich) erzeugen. Leider ist das Programm fehlerhaft und gibt manchmal größere Zahlen aus.

Die Entwickler haben vergessen, wie das Programm funktioniert, weil sie zu Faul waren, ihren Code zu kommentieren. Anstatt den Fehler auf die richtige Art und Weise zu beheben wurde deshalb die Entscheidung getroffen, den Ablauf des Programms mit Hilfe des GNU-Debuggers `gdb` so zu verändern, dass kein Fehler mehr auftritt.

(a) Informieren Sie sich selbstständig über folgende Befehle und dazugehörige Optionen für `gdb`:

```
backtrace break continue finish ignore info jump next return run step set var
```

Hierbei ist es ratsam, die Befehle an einem eigenen kleinen Programm auszuprobieren.

(b) Pfuschen Sie die Variablen des laufenden Roulette-Programms mit `gdb` so zurecht, dass die zu großen Zahlen stattdessen zur Zahl 36 werden und setzen Sie das Programm danach fort. Schreiben Sie Ihre Befehle in die Datei `gdb_eingabe.txt`. Sie dürfen die Datei `roulette.c` nicht verändern!

(c) Testen Sie ihre Befehle mit dem Skript `test.sh`. Das Skript wird `gdb` mit Ihren Befehlen aufrufen und in der Ausgabe überprüfen, ob das Programm richtig zu Ende lief.

**Aufgabe 4** *Check (Vorbereitungsaufgabe)*

Unter `04_check/string_suchen.c` finden Sie die Implementierung einer Funktion, die einen String in einem Text (ebenfalls ein String) suchen soll. Hierbei soll die Position vom gesuchten String im durchsuchten String zurückgegeben werden, falls der String gefunden wurde, oder $-1$, falls der String nicht gefunden wurde.

Zum Beispiel würde die Funktion für den Text „`Ein Entenbraten schmeckt gut.`“ und den Suchstring „`Enten`“ den Wert 4 zurückgeben. Für den Suchstring „`Truthahn`“ wäre das Ergebnis $-1$.

```
int string_suchen(char *text, char *string)
```

Leider ist diese Implementierung voller Fehler. Für diese Funktion gibt es zwar Tests, aber diese überprüfen nicht viele Eingaben, weshalb die Fehler nicht gefunden werden.

Kompilieren Sie die Tests und führen Sie sie aus, um sich davon zu überzeugen.

(a) Erweitern Sie die Tests, sodass alle Fehler gefunden werden.

(b) Beheben Sie die Fehler in `string_suchen`.

Iterieren Sie die Schritte wenn nötig, bis Sie sich sicher sind, dass die Funktion fehlerfrei ist.