Einführung in Unittests

- Zur jeder Unitestaufgabe gibt ein Datei mit Suffix .h, hier **beispiel.h**
- Ihre **.c** Datei **muss genau so heißen** wie die **.h** Datei und muss in .c Datei die Zeile **#include beispiel.h** beinhalten.
- Testdatei dürfen nicht geändert werden
- Alle Dateien einer Aufgabe sind in einem Ordner
- Ihre **.c** Datei **darf keine main()-Funktion** beinhalten, zum debuggen kann man extra Datei, welche eine main-Funktion beinhalten erstellen, diese muss **beispiel.h** Datei miteinbinden
- .ts Datei sind Test Dateien, hier **beispiel _tests.ts**
- Mit checkmk kann man die .ts Datei zu einer c. Datei umwandeln
  - **checkmk beispiel_tests.ts**
    checkmk nihmt .ts als parameter und gibt Ergebniss als der Standartausgabe stdout aus
  - **checkmk beispiel_tests.ts > beispiel_tests.c**
    wandelt die .ts Datei in eine neue .c Datei um
- Unittest-Framework check ist eine Library; übergibt dem Compiler Build-Flags
  - Parameter müssen indentifiziert werde, sind auf jedem Rechner anders
  - **pkg-config --cflag --libs check**
    gibt Build-Flags aus, diesmal sollte diese in einer Variable speicher:
    BUILDFLAGS="$(pkg-config --cflags --libs check)"
- Test mit gcc bauen:
  - **gcc -Wall -std=c99 -g beispiel_test.c beispiel_c -o tests $BUILDFLAGS _tests.ts** datei mit **.c** Datei linken, alle Warnungen und C99 Standart (**-Wall std=c99**), Debug-Information(**-g**), die Build-Flags müssen mit eingebunden werden (**$BUILDFLAGS**)
  - mit **./tests** kann das Programm wie gewöhnt ausgeführt werden
- Hinweise:
  - Reihenfolge der Build-Flags darf nicht geändert werden, außerdem müssen diese an der letzten Stelle stehen
  - kompilieren sie nicht die main.c bei erstellen des Tests
  - Variable **$BUILDFLAGS** wird nach der schießung des Shells nicht mehr verfügbar sein, Datei kann man persistent machen z.B. indem Sie diese in der Konfigurations-Datei .bashrc in ihrem Home-Verzeichnis zuweisen
- Weiter Informationen:
  - Dokumentation von check, Abschnitt 4.1 für Tests
  - Projektseite zu ckeck, der Abschnitt über Testfunktionen
- Alternativ: xargs
  - **pkg-config --cflags --libs check | xargs gcc -Wall -std=c99 -g beispiel.c beispiel_tests.c -o tests**
    erzeugung der Build-Flags und dem gcc anzuhängen
  - Pipe-Operator | leitet die Ausgabe eines Befehlt an anderen Befehl weiter
  - xarg nutzt Build-Flags als Argument für gcc
- Shell:
  - **gcc -Wall -std=c99 -g beispiel.c beispiel_tests.c -o tests $(pkg-config --cflags --libs check)**
    Ausgabe von pkg-config direkt als Argument zu nutzen

Valgrid & Debug

Verwendung von Strings tretten häufig Fehler auf, welche der Compiler nicht erkennt und die dann nur manchmal (aber nicht immer) zu Programmabstürzen führen. In solchen Fällen empfehlen wir das folgende Vorgehen:

- kompilieren mit **-Wall**, schaltet alle Warnungen an
- kompilieren mit **-g**, erlaubt es den Debugger zu nutzen
- mit **valgrind ./programmname**, wird valgrind mit eingebunden:
  - valgrind gibt Information zu Speicherzugriffsfehlern auf, mit der Zeile in welcher der Fehler aufgetretten ist.
  - häufige Fehler sind:
    - nicht genug Speicher für String/Array reserviert
    - Nullbyte am Ende des Strings überschrieben
    - keine Sprünge aufgrund nicht initialisierter Daten
- wenn valgrind nicht hilft, kann man mit debugger starten **gdb ./programmname**
  mit **run** startet man das Programm und mit **bt** (backtrace) kann man bei sich einzelne Funktionsaufrufe anschauen
- mit gcc können mit folgenden Optionen einige Speicherzugriffsfehler zur Laufzeit des Programms überprüfen:
  gcc -g -fsanitize=adress -fsanitize=undefined programm.c

GDB - Debugging

Alles hier gelistet kann man auf der Webseite finden:
http://www.unknownroad.com/rtfm/gdbtut/
https://www.tutorialspoint.com/gnu_debugger/gdb_quick_guide.htm

Da der Debugger offert viele Funktionalitäten. Hier ist die Liste der Am häufigsten verwendetsten Befehle:

b main	Puts a breakpoint at the beginning of the program
b	Puts a breakpoint at the current line
b N	Puts a breakpoint at line N
b +N	Puts a breakpoint N lines down from the current line
b fn	Puts a breakpoint at the beginning of function "fn"
d N	Deletes breakpoint number N
info break	list breakpoints
r	Runs the program until a breakpoint or error
c	Continues running the program until the next breakpoint or error
f	Runs until the current function is finished
s	Runs the next line of the program
s N	Runs the next N lines of the program

n	Like s, but it does not step into functions
u N	Runs until you get N lines in front of the current line
p var	Prints the current value of the variable "var"
bt	Prints a stack trace
u	Goes up a level in the stack
d	Goes down a level in the stack
q	Quits gdb

Initilise gdb:

Ermöglicht es einem das Programm zu debuggen und es nach möglichkeit leicht abzuändern. Um das Programm mit debugger zu öffnen muss es mit der Flage -g compiliert werden:

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

Ist das Programm mal compiliert kann man mit gdb aufrufen:

```
gdb programm
```

Man das Programm ausführen, mit **run** und dem Programm sogar argumente übergeben, wie eine Eingabe oder Optionen:

```
(gbd) run arg1 arg2 ....
```

Run ist ein Argument und kann durch andere wie **kill**,**help** und **list** ersetzt werden:
Das Programm kann mit **kill** wieder neu gestartet werden. Will man den Debugger verlassen gibt man **quit** ein. Beide Optionen muss man mit **yes/no** bestätigen. Mit **help** kann man eine Zusammenfassung alles Befehle und deren Funktionalität erhalten.

Watch Execution:

Mit Ctrl-C werden die exe beendet und mit continue wieder fortgesetzt. Mit **list** kann man die Fehler Zeile ausgeben lassen, hier gibt gcc zusätzlich einige der Zeilen bevor und nachher aus.

Will man schrittweise das Programm ablaufen kann man **next** und **step** nutzen. **Next** geht Funktionsweise durch den Code, wärend **step** in den Zeilenweise durhc Funktionen durchgeht.

Mit **print** kann man sich Variablen ausgeben lassen. Print wird der Varibalenname als Argument übergeben. Die führende Zahl ist ein Counter, welche zählt wie viele Variablen man sich angeschaut hat.

```
(gdb) print x
$1 = 900
```

Variablen können modifiziert werden mit **set**.

```
(gdb) set x = 3
(gdb) print x
$4 = 3
```

Mit call kann man Funktionen welche in das Programm verlinkt sind aufrufen, sowohl seien eigenen sowie Funktion auf Bibliotheken.
Zuletzt kann man eine Funktion beendet, mit **finish**, die Funktion wird bis zum Ende durchgerannt und kehrt dann zum Funktionsaufruf zurück. Hier wird auch der Funktion wiedergabe Wert mit ausgegeben.

```
(gdb) finish
Run till exit from #0  fun1 () at test.c:5
main (argc=1, argv=0xbffffaf4) at test.c:17
17        return 0;
Value returned is $1 = 1
```

Mit **return** kann die Funktion ebenfalls verlassen werden, jedoch anders als finish durchläuft man die Funktion nicht. Man verlässt die Funtion als ob man an der Stelle in der Funktion auf ein return gestoßen ist.

Back Tracking and Stack.

Mit **backtrace** kann man sich die Struktur der Aufrufe anschauen in der Funktion drunter sieht man wie die func2(), von der fun1(), welche wiederum von der main() aufgerufen wurde.

```
(gdb) backtrace
#0  func2 (x=30) at test.c:5
#1  0x80483e6 in func1 (a=30) at test.c:10
#2  0x8048414 in main (argc=1, argv=0xbffffaf4) at
test.c:19
#3  0x40037f5c in __libc_start_main () from
/lib/libc.so.6
(gdb)
```

Jeder Backtrace hat eine Nummer, man kann mit **frame** diesen Rahmen ausfrufen.
Um sich den Rahmen genau anschauen zu können hat man 3 Befehle. **info fram** gibt die Information über den Frame wieder. **info locals** gibt Information über die lokalen Variablen auf den Stackrahmen wieder, hier wird die Variable und Ihr Wert wiedergegeben. Zu letzt kann man mit info args die Lise alles Argumente ausgeben lassen.

```
(gdb) info frame
Stack level 2, frame at 0xbffffa8c:
eip = 0x8048414 in main (test.c:19); saved eip 0x40037f5c
called by frame at 0xbffffac8, caller of frame at
0xbffffa5c
source language c.
Arglist at 0xbffffa8c, args: argc=1, argv=0xbffffaf4
Locals at 0xbffffa8c, Previous frame's sp is 0x0
Saved registers:
ebp at 0xbffffa8c, eip at 0xbffffa90

(gdb) info locals
x = 30
```

```
s = 0x8048484 "Hello World!\n"

(gdb) info args
argc = 1
argv = (char **) 0xbffffaf4
```

Breakpoints

Mit **break** kann man eine Breakpoint setzen. Hier gibts du die Zeile an welche du einen Breakpoint haben möchtest als Argument. Bei mehreren Datei muss der Dateiname mit angegeben werden.

```
(gdb) break test.c:19
Breakpoint 2 at 0x80483f8: file test.c, line 19
```

Um ein Breakpoint in einer Funktion zu setzten gibt man statt Zeile die Funktion an welcher der Breakpoint gesetzt werden soll. Man kann auch temporäre Breaks einfügen mit **tbreak**.
Mit **info breakpoints** kann man sich alle Breakspoints ausgeben lassen.
Will man einen Breakspoint deactivieren so gibt man disable gefolgt vom der Breakpoint welchen man deactivieren möchte.

```
Num Type           Disp Enb Address    What
2   breakpoint     keep y   0x080483c3 in func2 at
test.c:5
(gdb) disable 2
(gdb) info breakpoints
Num Type           Disp Enb Address    What
2   breakpoint     keep n   0x080483c3 in func2 at
test.c:5
```

Zuletzt mit **ignore** kann man den Breakpoint x-mal ignorieren. Hier zu gibt man ignore gefolgt vom Point welchen man ignorieren möchte gefolgt von der Anzahl an male. Ein Breakspoint kann aber auch gelöscht werden mit **delete** gefolgt von der Breakpoint Nummer.

Watch Point

Mit **watch** condition kann eine Bedingung setzen ab welcher das Programm stoppt. Hier gilt jedoch das die Variable in dem Funktionsaufruf existiert. Ruft man watch nur mit einem Variablen name so wird das Programm angehalten, wenn sich der Wert der Varialen ändert.
Ähnlich wie mit Breakspoints kann man mit **list watchpoints** alle Watchpoints auflisten lassen.

Um eine Variable zu lesen kann man **rwatch** benutzen. Will man nur eine Variable auslesen so kann man **awatch** nutzen. Beide Variablen sind watch sehr ähnlich.
Wie Breakspoints können Watchpoint deactiviert werden oder gelöscht werden. Hier ist jedoch zu beachten, das **Watchpoints Breakpoints sind** und zum löschen man den Index in list breakpoints nutzen soll.

Mit mit **continue** kann man das Programm wieder fortsetzen. Will man an eine bestimmt Stelle im Code springen kann man **jump** benutzen. Jump springt ohne das Programm auszuführen an die gegeben Code stelle. Alternativ kann das pc register geändert werden um an eine bestimmt Stelle zu springen.

Advance gdb Features

Mit den Befehl x kann man sich den Speicher anschauen. Hier kann man entweder die Adresse aufrufen oder eine Variable

Mit x/s kann man sich eine String anschauen.

```
(gdb) x/s s
0x8048434 <_IO_stdin_used+4>:    "Hello World\n"
```

Mit x/c kann man sich den ersten Buchstaben im String anschauen.

```
(gdb) x/c s
0x8048434 <_IO_stdin_used+4>:   72 'H'
```

Mit x/4c kann man sich die ersten vier Buchstaben anschauen.

```
(gdb) x/4c s
0x8048434 <_IO_stdin_used+4>:   72 'H'  101 'e' 108 'l'
108 'l'
```

Mit x/t kann man sich die 32-bit der Variable anschauen.

```
(gdb) x/t s
0x8048434 <_IO_stdin_used+4>:
01101100011011000110010101001000
```

Mit x/3x kann man sich die ersten 24 byte der Variable in hex anschauen.

```
(gdb) x/3x s
0x8048434 <_IO_stdin_used+4>:   0x6c6c6548
0x6f57206f      0x0a646c72
```

Mit info register kann man sich den Inhalt der Register anschauen.

```
(gdb) info registers
eax            0x40123460       1074934880
ecx            0x1      1
edx            0x80483c0        134513600
ebx            0x40124bf4       1074940916
esp            0xbffffa74       0xbffffa74
ebp            0xbffffa8c       0xbffffa8c
esi            0x400165e4       1073833444
...
```

Mit **core core** kann man den Core dump aufrufen. Will man sich den Assembler Code anschauen so kann man dies mit **disassemble** tun. Wenn man durch den Assemble Code durchlaufen möchte so nutzt man **nexti** und **stepi**, diese sind identisch zu next und step.