

Kapitel 4: Das Fundament zur Linux-Systementwicklung

4: Werkzeuge zur Linux-Systementwicklung

In diesem Kapitel wollen wir uns kurz die folgenden Werkzeuge anschauen

- ▶ `gcc` und `clang` – C Compiler
- ▶ `valgrind` – Valgrind Debugger/Profiler
- ▶ `[l|s]trace` – A [library|system] call tracer
- ▶ `gdb` – GNU-Debugger
- ▶ `make` – Ein GNU-Tool zum Bauen von Programmen

4.1: GNU Compiler Collection (gcc)

Wichtige Compilerflags

`-O3` Optimierung

`-W` `-Wextra` `-Wall` Aktivierung alle Warnungen

`-Werror` Behandelt Warnungen wie Fehler

`-g` `-gdb3` Generiert Debug-Symbole

`-E` Führe nur den Präprozessor aus

`-S` Generiere Assembler-Code

`-c` Nur kompilieren nicht linkern

`-o file` Ausgabedatei

`-static` Generiere statische Binärdatei

Hallo Welt – Normales Kompilieren

```
1  #include <stdlib.h>
2  #include <stdio.h>
3
4  int main() {
5      puts("Hello_World!");
6      return EXIT_SUCCESS;
7  }
```

```
$ gcc -O3 -W -Wextra -Wall -Werror hw.c -o hw
```

```
$ ls -lah hw | cut -d " " -f 5,9
```

```
6,6K hw
```

```
$ file hw | cut -d "," -f 2,4
```

```
x86-64, dynamically linked
```

```
$ objdump -p hw | grep NEEDED
```

```
NEEDED                  libc.so.6
```

Hallo Welt – Kompilieren mit Debug-Symbolen

```
1  #include <stdlib.h>
2  #include <stdio.h>
3
4  int main() {
5      puts("Hello_World!");
6      return EXIT_SUCCESS;
7  }
```

```
$ gcc -ggdb3 -W -Wextra -Wall -Werror hw.c -o hw
$ ls -lah hw | cut -d " " -f 5,9
7,6K hw
$ file hw | cut -d ", " -f 2,4
x86-64, dynamically linked
$ objdump -p hw | grep NEEDED
NEEDED               libc.so.6
```

Hallo Welt – Statisch kompiliert

```
1 #include <stdlib.h>
2 #include <stdio.h>
3
4 int main() {
5     puts("Hello_World!");
6     return EXIT_SUCCESS;
7 }
```

```
$ gcc -O3 --static -W -Wall -Werror hw.c -o hw
$ ls -lah hw | cut -d " " -f 5,9
796K hw
$ file hw | cut -d "," -f 2,4
x86-64, statically linked
$ objdump -p hw | grep NEEDED
```

Anmerkungen

`-W -Wall -Wextra -Werror`: Verwenden Sie immer diese Compilerflags

`objdump`: Display information from object files.

`file`: Bestimmt den Dateitypen

- ▶ Verwenden Sie in der Regel `-ggdb3` oder `-O3`
- ▶ Statisch kompilierte Programme funktionieren auch ohne die `libc`
- ▶ Mittels `dietlibc` oder `ulibc` lassen sich auch schlanke statische Programme bauen
- ▶ Das Kommando `gcc` kann meist mit `clang` ausgetauscht werden

4.2: Das Dynamische Tracer-Duo: `ltrace` und `strace`

- ▶ `ltrace`
 - ▶ Zeigt alle Library-Calls (Bibliotheksaufrufe) an
 - ▶ Ausgabe: Funktionsnamen, Parameterlisten und Rückgabewerte
 - ▶ **Library-Call**: Aufruf einer Funktion welche Teil einer Programmbibliothek ist
- ▶ `strace`
 - ▶ Zeigt alle System-Calls (Systemaufrufe oder Syscall) an
 - ▶ Ausgabe: Funktionsnamen, Parameterlisten und Rückgabewerte
 - ▶ **Syscall**: Aufruf einer Funktion welche vom Kernel ausgeführt wird

Systemcall

- ▶ Viele Programme benötigen Funktionalität die nur der Kernel (Ring 0) bereitstellen kann
- ▶ Direkter Zugriff auf Hardware ist dem Ring 0 vorbehalten
- ▶ Glücklicherweise hat jeder Kernel eine API
- ▶ Der POSIX-Standard definiert diese Systemschnittstelle (**Achtung:** Windows unterstützt (noch?) kein POSIX)
- ▶ Bitte lesen Sie sich die folgenden Wikipedia Eintrag durch:
`https://de.wikipedia.org/wiki/Portable_Operating_System_Interface`
- ▶ Zu den Systemcalls gibt es auch Manpages (Abschnitt 2)
Beispiel: `$ man 2 write`
- ▶ Bitte lesen Sie sich die folgenden Wikipedia Eintrag durch:
`https://de.wikipedia.org/wiki/Systemaufruf`

Systemcall unter Linux Ausführen

▶ x86

- ▶ Assemblerbefehl: `int 0x80`
- ▶ Registerbelegung
 - ▶ Systemcall ID: `eax`
 - ▶ Parameter 1-6: `ebx, ecx, edx, esi, edi, ebp`

▶ x86_64

- ▶ Assemblerbefehl: `syscall`
- ▶ Registerbelegungen
 - ▶ Systemcall ID: `rax`
 - ▶ Parameter 1-6: `rdi, rsi, rdx, r10, r8, r9`

▶ Weiter Architekturen: `$ man 2 syscall`

Prozesseüberwachung mittels ptrace

```
#include <sys/ptrace.h>

long ptrace(enum __ptrace_request r, pid_t Pid,
            void *addr, void *data);
```

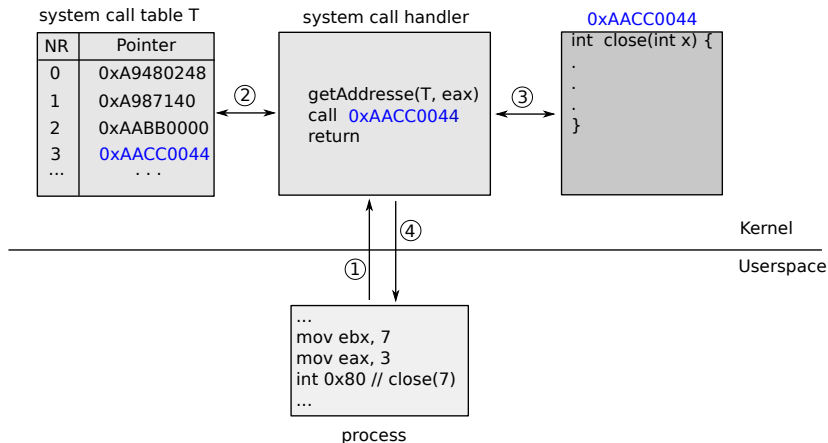
- ▶ Mit dem Systemcall `ptrace` lassen sich Prozesse überwachen.
- ▶ Es lassen sich Haltepunkte (engl. *break points*) setzen. Bei Erreichen eines solchen wird der überwachte Prozess angehalten.
- ▶ Mittels `ptrace` können auch Register- und Speicherinhalte gelesen oder beschrieben werden.
- ▶ Profiler und Debugger wie `ltrace`, `strace`, `gdb` und `valgrind` verwenden `ptrace` zur Prozessüberwachung.
- ▶ Weiter Informationen: `$ man 2 ptrace`

Systemcall unter Linux Ausführen

Die Systemcall ID ist in dem Headerfile `unistd_64.h` bzw. `unistd_32.h` definiert.

```
#define __NR_read 0
#define __NR_write 1
#define __NR_open 2
#define __NR_close 3
.
.
.
#define __NR_copy_file_range 326
#define __NR_preadv2 327
#define __NR_pwritev2 328
```

Systemcall Ausführen



strace

```
$ strace ./hw
execve("./hw", [ "./hw" ], [ /* 33 vars */ ]) = 0
...
open("/lib/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
read(3, ..., 832) = 832
...
close(3)
...
write(1, "Hello World!\n", 13Hello World!) = 13
exit_group(0)                                = ?
+++ exited with 0 +++
```

Die `strace`-Ausgabe beinhaltet auch das Laden des Prozesses und der benötigten Programmbibliotheken.

Bibliotheksaufrufe

- ▶ Implementation der Funktion befindet sich in einer anderen Binärdatei.
- ▶ Beim Starten des Programmes werden die benötigten Funktionen aus den entsprechenden Bibliotheken nachgeladen
- ▶ Programme sind in der Regel von Bibliotheken abhängig
- ▶ Dynamisch gelinkte Programme (Default) hängen von der Standard-C-Bibliothek `libc` ab.
- ▶ Der Befehl `$ objdump -p hw | grep NEEDED` zeigt alle Abhängigkeiten des Programms `hw` an.
- ▶ Zu den Standard-Bibliotheksaufrufen gibt es auch Manpages (Abschnitt 3). **Beispiel:** `$ man 3 puts`
- ▶ Bitte lesen Sie sich den folgenden Wikipedia-Eintrag durch:
<https://de.wikipedia.org/wiki/Programmbibliothek>

ltrace

```
$ ltrace ./hw
__libc_start_main(0x400520, 1, ...
puts("Hello World!"Hello World!)  = 13
+++ exited (status 0) +++
```

Die `ltrace`-Ausgabe verrät:

Die Funktion `main` ruft die Bibliotheksfunktion `puts` mit dem Parameter "Hello World!" auf. Der Rückgabewert ist 13. Anschließend wird das Programm erfolgreich beendet.

Merke: Mit der Option `-l library_pattern` werden nur Bibliotheksaufrufe von ausgewählten Bibliotheken angezeigt.

Beispiel: `$ ltrace -l libc* ./hw`

4.3: Der Valgrind Debugger/Profiler

- ▶ Valgrind ist der Haupteingang nach Valhalla und eine Toolsammlung zum Debuggen und Profilen
- ▶ Valgrind beobachtet einen Prozess um Laufzeitfehler zu erkennen
- ▶ Schlanke GUI: `alleyoop`
- ▶ Für die Vorlesung beschränken wir uns auf `memcheck`
- ▶ `Memcheck` erkennt folgende Fehler
 - ▶ Zugriff auf nicht-reservierten Speicher
 - ▶ Probleme mit nicht-initialisierten Variablen
 - ▶ Speicherlecks (engl. *memory leaks*)
 - ▶ *Double frees* und *mismatched frees*
 - ▶ Aufruf von `memcpy()` mit überlappenden Speicherbereichen

memerrors.c

```
1  #include <stdlib.h>
2
3  void foo(void) {
4      int* x = malloc(23 * sizeof(int));
5      x[23] = 0;
6  }
7
8  int main(void) {
9      foo();
10     return EXIT_SUCCESS;
11 }
```

Das Programm hat 2 Fehler:

- ▶ Allozierter Speicher wird nicht wieder freigegeben
- ▶ Es wird auf nicht-initialisierten Speicher zugegriffen: `x[23]`

```
$ gcc -ggdb3 -W -Wextra -Wall -Werror memerrors.c -o memerr
```



Valgrind Beispiel memerrors

```
valgrind --leak-check=full -v ./memerrors
...
==21746== ERROR SUMMARY: 2 errors from 2 contexts (suppressed: 0 from 0)
==21746==
==21746== 1 errors in context 1 of 2:
==21746== Invalid write of size 4
==21746==      at 0x400504: foo (memerrors.c:5)
==21746==      by 0x400515: main (memerrors.c:9)
==21746== Address 0x51da09c is 0 bytes after a block of size 92 alloc'd
==21746==      at 0x4C29C0F: malloc (vg_replace_malloc.c:299)
==21746==      by 0x4004F7: foo (memerrors.c:4)
==21746==      by 0x400515: main (memerrors.c:9)
==21746==
==21746== ERROR SUMMARY: 2 errors from 2 contexts (suppressed: 0 from 0)
```

4.4: GDB – GNU Debugger

- ▶ Debugger für verschiedene Programmiersprachen wie C/C++
- ▶ Primäres Ziel: Ursachenbestimmung von Laufzeitfehlern
- ▶ Ermöglicht es ein Programm zu einem bestimmten Zeitpunkt anzuhalten und zu untersuchen
- ▶ Ermöglicht die Analyse von Speicherzugriffsfehlern (engl. *segmentations faults*)
- ▶ Beim Kompilieren mittels `gcc` oder `clang` sollte die Option `-ggdb3` gesetzt sein
- ▶ Eine GDB-GUI: `xxgdb`
- ▶ GDB-Handbuch: `https://sourceware.org/gdb/current/onlinedocs/gdb/`

GDB – Survival Guide

- ▶ GDB ist ein CLI (Command Line Interface)
- ▶ Das Kommando `$ gdb` start eine interaktive *Shell*
- ▶ GDB unterstützt Autovervollständigung ()
- ▶ Wichtige Kommandos
 - ▶ `quit`: Beendet das Programm
 - ▶ `help [command]`: Hilfe zu einem Kommando
 - ▶ `apropos <word>`: Kommandosuche
 - ▶ `file <prog>`: Lädt ein Programm
 - ▶ `run`: Startet das Programm
 - ▶ : Wiederhole den letzten Befehl

GDB Fallbeispiel – segfault.c

```
1  #include <stdlib.h>
2  void foo() {
3      int a = 21;
4      int *x = NULL;
5      a += a;
6      *x = a;
7  }
8
9  int main(void) {
10     foo();
11     return EXIT_SUCCESS;
12 }
```

```
$ gcc -W -Wextra -Wall -Werror -g segfault.c -o segfault
```

```
$ gdb ./segfault
```

```
(gdb) run
```

```
...
```

```
Program received signal SIGSEGV, Segmentation fault.
```

```
0x00000000004004c8 in foo() at segfault.c:7
```

```
7    *x = a;
```

GDB Aussagekraft – segfault.c

```
$ gcc -W -Wextra -Wall -Werror -g segfault.c -o segfault
$ gdb ./segfault
(gdb) run
...
Program received signal SIGSEGV, Segmentation fault.
0x00000000004004c8 in foo() at segfault.c:7
7      *x = a;
```

- ▶ Das Programm crasht bei dem Befehl `*x = a;`
- ▶ Der Befehl steht in der Datei `segfault.c` in Zeile 7
- ▶ Der Crash passiert in der Methode `foo()`
- ▶ Diese Information helfen bei der Fehlersuche

segfault.c – Überwachung der Variable *a*

- ▶ Gedanke: Vielleicht liegt das Problem ja bei *a*.
- ▶ Stoppe das Programm bei dem Aufruf der Funktion `foo()`
- ▶ Dann überwache die Variable *a*.

(→ GDB Demo)

```
(gdb) break foo
```

```
(gdb) run
```

```
(gdb) watch a
```

```
(gdb) continue
```


segfault.c – Durch eine Funktion steppen

- ▶ Wir wissen: Die Funktion `foo()` verursacht das Problem
- ▶ Unterbreche das Programm bei Eintritt in die Funktion `foo()`
- ▶ Steppe zeilenweise durch die Funktion `foo()`

(→ Demo)

```
(gdb) break foo
(gdb) run
(gdb) step
(gdb)
(gdb)
(gdb)
(gdb)
```

segfault.c – Durch eine Funktion steppen

- ▶ Wir wissen: Befehl in Zeile `segfault.c:7` verursacht das Problem
- ▶ Unterbreche das Programm an dieser Stelle
- ▶ Schaue dir die lokalen Variablen `a` und `x` an

(→ Demo)

```
(gdb) break segfault.c:7
(gdb) run
(gdb) print a
(gdb) print x
(gdb) next
```

Breakpoints

Breakpoints sind Haltepunkte die gesetzt werden können

- ▶ **break: Breakpoint setzen**
 - ▶ `break function`
 - ▶ `break file.c:42`
 - ▶ `break file.c:6 if i >= ARRAYSIZE`
- ▶ **info breakpoints: Liste der Breakpoints**
- ▶ `delete <breakpoint id>`

Ausführung

- ▶ `run`: Startet die Ausführung
- ▶ `continue`: Nimm Ausführung nach Unterbrechung wieder auf
- ▶ `finish`: Ausführung bis zum Funktionsende
- ▶ `step`: Gehe einen Schritt weiter
- ▶ `step 5`: Gehe 5 Schritte weiter
- ▶ `next`: Wie `step`, aber Funktionsaufruf zählt als ein Schritt

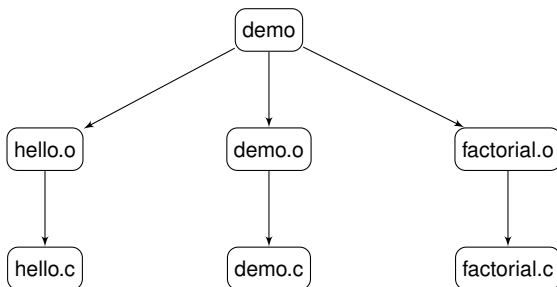
Informationen über Variablen

- ▶ `info variables`: Anzeige aller globalen Variablen
- ▶ `info locals`: Anzeige aller lokalen Variablen
- ▶ `info args`: Liste der Argumente auf dem Stack-Frame
- ▶ `info registers`: Ausgabe regulärer Register
- ▶ `backtrace`: Ausgabe des Stacktraces

Ausgabe

- ▶ `print var`: Gibt Wert einer Variablen aus
- ▶ `print/x var`: Gibt Wert einer Variablen als Hex-String aus
- ▶ `print e->var`: Gibt Wert einer Struct-Variablen aus
- ▶ `print (*e).var`: Gibt Wert einer Struct-Variablen aus
- ▶ `print (*e)`: Gibt alle Werte eines Structs aus

4.5: make – Bauanleitung für Programme



- ▶ Das Compilieren von Programmen ist oftmals kompliziert
- ▶ Von Hand bauen ist oft sehr mühselig
- ▶ Daher gibt es Programme wie `make` mit denen sich der Build-Prozess automatisieren lässt
- ▶ Einige Alternativen zu `make`: `SCons`, `CMake` und `Gradle`

Eine Demoanwendung – Teil 1

demo.c

```
1  #include <stdlib.h>
2  #include <stdio.h>
3  #include "functions.h"
4
5  int main(){
6      print_hello();
7      printf("The_factorial_of_5_is_%u\n", factorial(5));
8      return EXIT_SUCCESS;
9  }
```

functions.h

```
1  # pragma once
2  void print_hello();
3  unsigned int factorial(unsigned int a);
```


Eine Demoanwendung – Teil 2

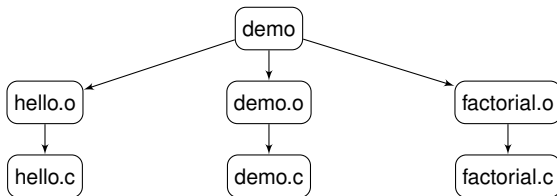
hello.c

```
1  #include <stdio.h>
2  #include "functions.h"
3  void print_hello() {
4      puts("Hello_World!");
5  }
```

factorial.c

```
1  #include "functions.h"
2
3  unsigned int factorial(unsigned int a) {
4      if(a<2) return a;
5      else   return(a * factorial(a-1));
6  }
```

Manuelles Bauen




- ▶ **Annahme:** Alle Quellendatei der Demoanwendung befinden sich in einem Verzeichnis (z.B. `src`).
- ▶ Bei dieser Demo ist der manuelle Buildvorgang noch einfach
`$ gcc -W -Wextra -Wall demo.c hello.c factorial.c -`
- ▶ Das manuelle Bauen skaliert aber nicht.
- ▶ Stellen Sie sich ein größeres Softwareprojekt wie z.B. Mozilla-Firefox mit hunderten bzw. tausenden von C-Dateien, vor.

Makefile Regeln

Syntax einer Makefile-Regel

target: Zutaten

 Rezept

 ...

 ...

Variablen

- ▶ Zuweisung: `FOO = bar`
- ▶ Lesender Zugriff (Referenzierung von Variablen)
 - ▶ `$(FOO)` oder
 - ▶ `${FOO}`
- ▶ Funktionsweise des Referenzieren: strikte Textersetzung.
- ▶ Referenz wird durch Variableninhalt ersetzt.

Demoanwendung – Makefile v1

Makefile

```
1  WARNFLAGS = -W -Wall -Werror
2  OPTFLAGS = -O3
3  CFLAGS= $(WARNFLAGS) $(OPTFLAGS)
4
5  demo:
6      $(CC) $(CFLAGS) demo.c hello.c factorial.c -o $@
7
8  clean:
9      rm -f *~ *.o demo
```

- ▶ `$@` steht für das Target (in unserem Fall `demo`)
- ▶ Der Befehl `$ make` baut die Demoanwendung
- ▶ Der Befehl `$ make clean` löscht die Demoanwendung

Vordefinierte Variablen

- ▶ **RM**: Steht für `rm -f`.
- ▶ **CC**: Standard C-Compiler
- ▶ **CPP**: Standard C++-Compiler
- ▶ **CURDIR**: Aktuelles Verzeichnis
- ▶ **OUTPUT_OPTION**: Steht für `-o $@`
- ▶ **COMPILE.c**: Steht für C-Datei Compilieren.
- ▶ Das Kommando `$ make -p` gibt die Liste aller vordefinierten Regeln und Variablen aus.

Gängige Variablen

- ▶ `CFLAGS`: Compiler-Flags für den C-Compiler
- ▶ `CPPFLAGS`: Compiler-Flags für den C++-Compiler
- ▶ `OBJC`: Kompilierte C-Dateien
- ▶ `LDLIBS`: Bibliotheken die für das Linken benötigt werden.
- ▶ `ODIR`: Object directory

Automatic Variables

Es gibt **Automatic Variables** die ohne Klammern referenziert werden.

- ▶ `$@`: Target
- ▶ `$<`: Erste Zutat
- ▶ `$^`: Menge aller Zutaten
- ▶ `$+`: Liste aller Zutaten
- ▶ `$?`: Liste aller Zutaten die neuer sind als das Target.

Demoanwendung – Makefile v2

```
1  CFLAGS = -W -Wall -Werror -O3
2  OBJS = demo.o hello.o factorial.o
3
4  demo: $(OBJS)
5      $(CC) $(CFLAGS) $(OBJS) -o $@
6
7  %.o: %.c
8      $(CC) $(CFLAGS) -c $<
9
10 clean:
11     $(RM) *~ *.o demo
```

► `%.o: %.c`

Rezept wie ich aus einer Datei mit der Endung `.c` eine Datei mit der Endung `.o` erstelle.

- Die Regel wird angewandt, falls eine entsprechende `.o` Datei benötigt wird, bzw. eine neuere `.c` Datei vorliegt.

Vordefinierte Regeln

```
%.o: %.c
```

```
# Auszuführende Regel (eingebaut):
```

```
$(COMPILE.c) $(OUTPUT_OPTION) $<
```

```
%.o: %.cpp
```

```
# Auszuführende Regel (eingebaut):
```

```
$(COMPILE.cpp) $(OUTPUT_OPTION) $<
```

```
%.o: %.s
```

```
# Auszuführende Regel (eingebaut):
```

```
$(COMPILE.s) -o $@ $<
```

Pseudo-Targets

- ▶ Targets werden nur gebaut, falls diese noch nicht existieren.
- ▶ Pseudo-Targets sollen immer gebaut werden.
- ▶ Alle Zutaten des Spezial-Targets `.PHONY` sind Pseudo-Targets.
- ▶ **Beispiel:** `.PHONY: clean`

Demoanwendung – Makefile final

```
1  WARNFLAGS = -W -Wall -Werror
2  OPTFLAGS = -O3
3  DEBUGFLAGS = -ggdb3
4  CFLAGS= $(WARNFLAGS)
5  OBJ= demo.o hello.o factorial.o
6
7  ifdef DEBUG
8      CFLAGS += $(DEBUGFLAGS)
9  else
10     CFLAGS += $(OPTFLAGS)
11 endif
12
13 all: demo
14
15 demo: $(OBJ)
16     $(CC) $(CFLAGS) $^ $(OUTPUT_OPTION)
17
18 .PHONY: clean
19
20 clean:
21     $(RM) -f *~ *.o demo
```

Anmerkungen zum if-else-Verzweigung

```
...  
ifdef DEBUG  
CFLAGS += $(DEBUGFLAGS)  
else  
CFLAGS += $(OPTFLAGS)  
endif  
...
```

- ▶ `$ make`
Erstellung des regulären Programms
- ▶ `$ make DEBUG=1`
Erstellung der Debug-Variante des Programms

Zusammenfassung

Sie sollten in der Lage sein...

- ▶ ... ein C-Programm zu kompilieren.
- ▶ ... mittels `ltrace` und `strace` die Funktionsweise eines einfachen Programmes zu rekonstruieren.
- ▶ ... mittels Valgrind lassen sich Programmierfehler wie Speicherlecks finden.
- ▶ ... mit Hilfe des GNU-Debuggers Laufzeitfehler zu finden und diese im Anschluss zu beheben.
- ▶ ... einfache Makefiles zu erstellen.