

# Systemprogrammierung

## Teil 8: Werkzeuge

Programmerstellung, Fehlersuche

# Werkzeuge: Einsatzgebiete

---

## Erstellen von Programmen

- **Bearbeiten** von Quellcode  
Schreiben von neuem Quellcode  
Ändern von vorhandenem Quellcode
- **Transformieren** von Quellcode in ausführbaren Code  
je nach Programmiersprache mehrere Transformationsschritte erforderlich  
bei aus vielen Teilen bestehenden Programmen Wiederholen der Transformationsschritte pro Programmteil erforderlich  
*bei C: Präprozessor, Compiler, Linker*

## Prüfen von Programmen

- **Testen** der funktionalen und nicht-funktionalen Eigenschaften
- **Fehlersuche**

# Werkzeuge: Nutzen

---

- Ermöglichung

Werkzeug Texteditor unverzichtbar zum Bearbeiten von Quellcode (z.B. *vSCode*)

Werkzeug Compiler unverzichtbar zum Transformieren von Quellcode (z.B. *gcc*)

- Automatisierung

bei aus vielen Teilen bestehenden Programmen sehr viele Arbeitsschritte, die Arbeitsschritte automatisch veranlassen (z.B. mit *Shell-Script* oder *make*).



- Optimierung

bei Programmänderungen nur die notwendigen Arbeitsschritte durchführen, unnötige Arbeitsschritte automatisch weglassen (z.B. mit *Werkzeug make*)

- Qualitätssicherung

Mängel im Quellcode und bei Transformationsschritten entdecken / vermeiden (z.B. *statische Codeanalyse mit gcc und cppcheck*). 

# Bearbeiten von Quellcode: Formatierung

---

**astyle** – ein "Beautifler" für C / C++ / C# / Java-Quellcode

Aufruf-Möglichkeiten:

**astyle [Optionen] Quelldatei ...**

*ursprünglicher Code wird nach  
Quelldatei.orig gerettet.*

**astyle [Optionen] < hässliche\_Datei > verschönerte\_Datei**

- Optionen: 

Festlegung des Formatierungsstils

(Einrückung und Klammerung von Blöcken, Platzierung von Zwischenraum, ...):

*z.B. --style=ansi Einrückungs- und Klammerungsstil nach ANSI*

*z.B. -p Leerstellen um Operatoren herum*

Festlegungen der Quellsprache (bei Aufruf mit Dateiumlenkung):

*z.B. --mode=c*

- Funktionsweise:

korrigiert die Formatierung in den angegebenen Quelldateien

# Bearbeiten von Quellcode: Suchen und Vergleichen

---

Bearbeiten von Quellcode bedeutet vor allem korrigieren, ändern und erweitern. Dazu müssen die relevanten Stellen im vorhandenen Code gefunden werden.

- Dateien suchen mit den Linux-Kommandos find und grep:

```
find original/ -mtime 0 -name *.c -print
```

*liefert die Namen aller .c-Quelldateien im Verzeichnisbaum unter original/, die innerhalb der letzten 24 Stunden geändert wurden*

```
grep -rI "gruessen()" original/
```

*liefert die Namen aller Dateien im Verzeichnisbaum unter original/, die die Zeichenkette gruessen() enthalten*

- Dateien und Dateibäume vergleichen mit dem Linux-Kommando diff:

```
diff -rq original/ backup/
```

*liefert die Namen aller Dateien, die sich inhaltlich unterscheiden oder nur in einem der beiden Verzeichnisbäume vorhanden sind*



```
diff original/hallo.c backup/hallo.c
```

*liefert alle Zeilen aus den beiden Dateien, die sich unterscheiden*

# Transformieren von C-Quellcode: gcc

---

**gcc** – der GNU Präprozessor / Compiler / Assembler / Linker für C

- Aufruf:

**gcc [Option ...] Eingabedatei ...** 

- Optionen:

**[-E | -S | -c]** *Transformationsschritte einschränken*

**[-Dmacro [=definition] ...] [-Umacro ...] [-I dir ...]** *Präprozessor steuern*

**[-std=standard] [-pedantic] [-Wwarn ...]** *"Strenge" des Compilers steuern*

**[-g] [-pg]** *Debuggen und Vermessen vorbereiten*

**[-Olevel]** *Code-Optimierung steuern*

**[-L dir ...] [-l name ...]** *Linker steuern*

**[-o outfile]** *Name der Ergebnisdatei angeben*

... *insgesamt über 1000 Optionen, ca. 650 Seiten Dokumentation*

- empfohlene Optionen zur **Qualitätssicherung** von C-Quellcode:

**-Wall -Wextra -Werror -std=c11 -pedantic**

*vor potenziellen  
Fehlern warnen*

*Warnungen als  
Fehler werten*

*Einhaltung des Sprachstandards  
überwachen (hier ISO C11)*

# Übersetzungseinheiten: Beispiel

---

Einfaches C-Programm mit zwei Übersetzungseinheiten:

```
/* hallo.c */  
#include "gruss.h"  
int main(void)  
{  
    gruessen();  
    return 0;  
}
```

```
/* gruss.h */  
#ifndef GRUSS_H  
#define GRUSS_H  
void gruessen(void);  
#endif
```

```
/* gruss.c */  
#include "gruss.h"  
#include <stdio.h>  
void gruessen(void)  
{  
    printf("Hallo\n");  
}
```

- Global sichtbare Namen in der **Header-Datei** (Endung **.h**) deklarieren.
- Header-Datei per **#include** in die Implementierungs-Datei (Endung **.c**) kopieren.

# Übersetzungseinheiten: Compiler und Linker-Aufrufe

---

Compiler/Linker-Aufrufe bei Programmen mit mehreren Übersetzungseinheiten: 

- jede Übersetzungseinheit getrennt übersetzen:

```
gcc -c -I. hallo.c
```

```
gcc -c -I. gruss.c
```

*Der Präprozessor kopiert `gruss.h` jeweils in `hallo.c` bzw. `gruss.c` hinein.*

*Option(en) `-I` geben an, wo Header-Dateien anderer Übersetzungseinheiten liegen.*

- dann den Objektcode der Übersetzungseinheiten (Endung `.o`) binden:

```
gcc hallo.o gruss.o -o hallo
```

*Das ausführbare Programm nennt man üblicherweise so wie die Übersetzungseinheit mit dem Hauptprogramm `main`.*



# Transformieren von Quellcode: Probleme

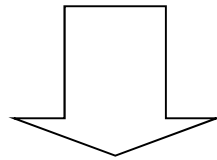
---

Manueller Aufruf von Compiler und Linker zu **aufwendig**:

- bei Programmen mit vielen Übersetzungseinheiten viele Aufrufe notwendig
- eventuell viele Optionen pro Aufruf

Manueller Aufruf von Compiler und Linker zu **fehlerträchtig**:

- nach Programmänderungen Vergessen notwendiger Aufrufe
- ungünstige oder falsche Optionen bei den Aufrufen



Abhilfe durch  
**Automatisierung**

# Transformieren von Quellcode: Automatisierung

---

**Kommandoprozeduren** automatisieren die Programmerstellung:

- zur Programmerstellung erforderliche Kommandofolge in eine Datei schreiben
- die Datei ausführen, um die Kommandofolge zu wiederholen

*Auch nach kleinen Programmänderungen werden alle Kommandos ausgeführt. Das kann bei Programmen mit vielen Übersetzungseinheiten sehr lange dauern.*

**Build-Werkzeuge** automatisieren und optimieren die Programmerstellung:

- die Abhängigkeiten zwischen den zu erstellenden Endergebnissen, Zwischenergebnissen und Quellen sowie die erforderlichen Kommandos in einem Bauplan festhalten
- für den Bauplan das Build-Werkzeug aufrufen, um Zwischen- und Endergebnisse inkrementell erstellen bzw. aktualisieren zu lassen

*Es werden immer nur die laut Bauplan erforderlichen Kommandos ausgeführt.*

# Kommandoprozedur: Linux Shell-Script (1)

---

- eine **Linux-Shell** ist ein Programm, mit dem Benutzer Linux über Kommandos bedienen können (*Kommandointerpretierer*)

Es gibt verschiedene Implementierungen, die wichtigsten unter Linux sind:

**sh** Bourne Shell (für Kommandoprozeduren üblich) 

**bash** Bourne Again Shell (Standard für die interaktive Benutzung)

- ein **Shell-Script** ist eine Datei mit einer Folge von Linux-Kommandos:

```
#!/bin/sh   
gcc -c hallo.c  
gcc -c gruss.c  
gcc hallo.o gruss.o -o hallo
```

- Shell-Script ausführen:

**sh *Dateiname***

**./*Dateiname***

*für die zweite Variante muss bei der Datei  
das Ausführungsrecht gesetzt sein*

## Kommandoprozedur: Linux Shell-Script (2)

---

- die Bourne-Shell kennt auch Variablen, Verzweigungen und Schleifen:

```
#!/bin/sh
```

```
for s in hallo.c gruss.c ; do
```

```
    compile_command="gcc -c $s"
```

```
    echo $compile_command
```

```
    eval $compile_command
```

```
    if [ $? -ne 0 ] ; then
```

```
        echo build failed
```

```
        exit 1
```

```
    fi
```

```
done
```

```
link_command="gcc -o hallo hallo.o gruss.o"
```

```
echo $link_command
```

```
eval $link_command
```

```
if [ $? -ne 0 ] ; then
```

```
    echo build failed
```

```
    exit 1
```

```
fi
```

```
echo build successful
```

*# Variable mit Initialisierung *

*# Wert der Variablen ausgeben *

*# Wert der Variablen als Kommando ausführen*

*# Rückgabewert des Kommandos prüfen *

# Build-Werkzeug: GNU make

---

**make** – das Build-Programm unter Linux (*Unix, ...*)

Aufruf:

```
make [-f Bauplan] [Ziel] ...
```

- fehlt die Option `-f Bauplan`, wird **makefile** oder **Makefile** verwendet  
*Üblicherweise wird der Bauplan **Makefile** genannt,  
in speziellen Fällen werden auch Dateinamen mit Endung **.mak** oder **.mk** verwendet*
- **Ziel** ist eine zu erstellende Datei oder der Name einer Regel  
*fehlt das **Ziel**, wird das im Bauplan als erstes genannte Ziel bearbeitet,  
üblicherweise heißt das erste Ziel im Bauplan **all***
- sind mehrere **Ziele** angegeben, werden diese nacheinander bearbeitet

# GNU make: Beispiel (1)

---

- einfacher Bauplan für das Programm hallo:

```
# Makefile 
```

```
hallo: hallo.o gruss.o  
    gcc hallo.o gruss.o -o hallo
```

```
hallo.o: hallo.c gruss.h  
    gcc -c hallo.c
```

```
gruss.o: gruss.c gruss.h  
    gcc -c gruss.c
```

*Abhängigkeit  
(hallo abhängig von zwei Objekdateien)*

  
*Kommando  
(erzeugt hallo aus  
zwei Objekdateien)*

*Tabulator vor dem Kommando nicht vergessen *

- Aufruf zum Erstellen bzw. Aktualisieren des Programms:

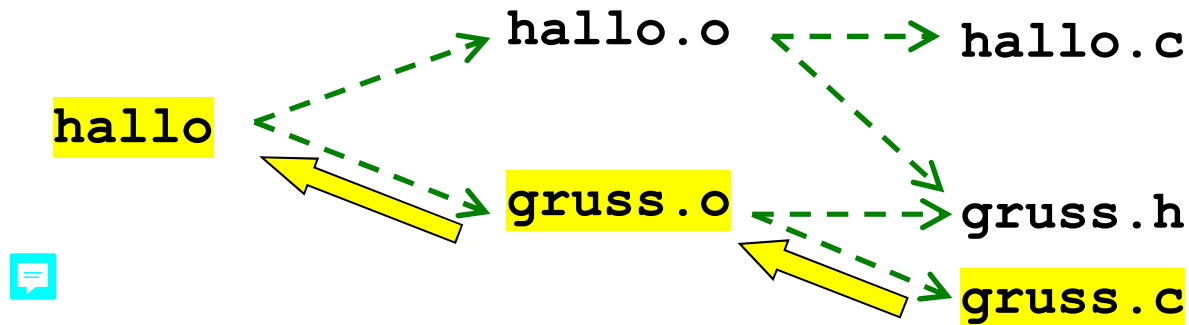
```
make -f Makefile hallo
```

```
make # tut das gleiche, weil Makefile Standardname und hallo erstes Ziel ist
```

# GNU make: Beispiel (2)

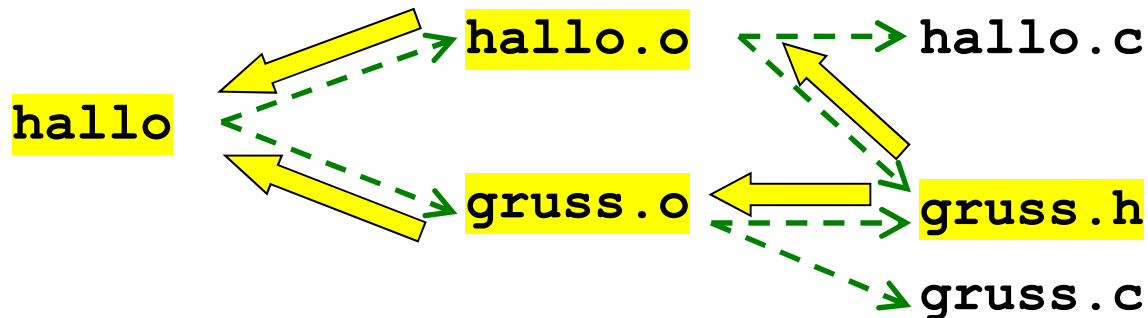
Abhängigkeiten (--->) steuern das **inkrementelle Erstellen** (←):

- Aufruf nach Änderung von `gruss.c`



*hallo.o wird nicht neu erstellt, weil unabhängig von gruss.c*

- Aufruf nach Änderung von `gruss.h`



*alles wird neu erstellt, weil abhängig von gruss.h*

# Makefile: Regeln

---


- explizite Regeln:

*Ziel* `hallo.o: hallo.c` *abhängig von* *Quelle*  
`gcc -c hallo.c -o hallo.o`  
*Kommando (muss mit Tabulator eingerückt sein)*

- Abhängigkeitsregeln sind explizite Regeln ohne Kommando:

```
hallo.o: hallo.c
```

- Musterregeln liefern das Kommando zu gleichartigen Abhängigkeitsregeln:

*Ziel mit % als Platzhalter für z.B. hallo* `%.o: %.c` *Quelle mit % als Platzhalter für z.B. hallo*  
`gcc -c $< -o $@`  
*\$< ist die Quelle, auf die die Regel angewendet wird*  *\$@ ist das Ziel, auf das die Regel angewendet wird*




# Makefile: explizite Regeln

---

```
Ziel ...: Quelle ...  
<TAB>Kommando  
<TAB>...
```

Anwendung auf Dateien:

- **Ziel** ist eine Datei, die mit der Regel erzeugt wird.  
Meist ein Ziel pro Regel, es sind aber auch mehrere erlaubt.
- **Quelle** ist eine Datei, die zum Erstellen des Ziels gebraucht wird.  
Keine, eine oder viele Quellen pro Regel.
- **Kommando** ist ein Befehl, der Zieldatei(en) aus Quelldatei(en) erzeugt.  
 Meist nur ein Kommando pro Regel, aber auch mehrere Kommandos oder komplizierte Kommandos in Shell-Skript-Syntax möglich.  
Liefert ein Kommando einen Fehlerstatus, beendet sich make automatisch.

# Makefile: Abhängigkeitsregeln

---

- Eine Abhängigkeitsregel ist eine explizite Regel ohne Kommando:

```
hallo.o: hallo.c gruss.h
gruss.o: gruss.c gruss.h
```

*Abhängigkeitsregeln sind die in der Praxis am häufigsten verwendete Form der expliziten Regel*

- Abhängigkeitsregeln brauchen zur Ergänzung Musterregeln, die die Kommandos festlegen, z.B.:

```
% .o: % .c
    gcc -c $< -o $@
```

*die automatische Variable  
\$< enthält den Namen der Quelldatei,  
\$@ den Namen der Zieldatei*

- Abhängigkeitsregeln kann der `gcc` automatisch aus den C-Quellen erzeugen, indem er die `#include`-Anweisungen auswertet (analog der `g++`):

```
gcc -MM hallo.c gruss.c > depend
```

*schreibt die Regeln mittels Umlenkung der Standardausgabe in die Datei `depend`, die Datei `depend` kann dann per `include` in das Makefile integriert werden*

# Makefile: Pseudoziele

---

- Ein **Pseudoziel** ist keine Datei, sondern ein beliebiger Name, der nur dazu dient, bestimmte Arbeitsschritte gezielt aufrufbar zu machen:

```
make Pseudoziel
```

*engl. unecht*

- Aufzählung der Pseudoziele im Makefile mit einer **.PHONY**-Regel:

```
.PHONY: all clean install uninstall
```

*die Pseudoziele `all`, `clean`, `install`, `uninstall` haben sich als Quasistandard eingebürgert*

- Die **a11**-Regel zählt alle Endergebnisse des Makefiles auf:

```
a11: hallo
```

*Die `a11`-Regel sollte immer die erste Regel im Makefile sein!*

- Die **clean**-Regel löscht alle Zwischen- und Endergebnisse, die mit dem Pseudoziel `a11` erzeugt werden:

```
clean:   
rm -f hallo hallo.o gruss.o 
```

# Makefile: Musterregeln

---

Eine **Musterregel** ist eine explizite Regel, bei der Quelle und Ziel ein %-Zeichen als Platzhalter für eine beliebige Zeichenkette enthalten:

- das Kommando einer Musterregel wird für Dateien ausgeführt, deren Name dem Muster entspricht und für die es keine explizite Regel gibt

GNU make hat für die wichtigsten Dateitypen **vordefinierte Musterregeln**, z.B.:

- übersetzen und binden eines beliebigen C-Programms mit nur einer Quelldatei:

```
%: %.c
    $(CC) $(CPPFLAGS) $(CFLAGS) $(LDFLAGS) -o $@ $<
```



automatische Variablen  
für Ziel und Quelle

- übersetzen einer beliebigen C-Quelle:

```
%.o: %.c
    $(CC) $(CPPFLAGS) $(CFLAGS) -c $<
```

*Die Kommandos sind mit Variablen definiert, um sie leicht an unterschiedliche Plattformen anpassen zu können:*

*cc enthält den Namen des C-Compilers, CPPFLAGS die Präprozessor-Optionen, CFLAGS die Compiler-Optionen und LDFLAGS die Linker-Optionen (bei C++ Variablen CXX und CXXFLAGS statt CC und CFLAGS)*

# Makefile: Variablen (1)

---

Mit **Variablen** können häufig wiederkehrende Bestandteile von Makefiles zusammengefasst und parametrisiert werden.

- Variablendefinition:

**`Variable = Wert`** oder mehrzeilig

**`Variable = \  
Wert \  
Fortsetzung`**



Zeilenwechsel  
müssen mit \  
maskiert werden

Variablenname üblicherweise in Großbuchstaben  
Wert ist eine beliebige Zeichenkette

- Variablenbenutzung:

**`$(Variable)`**

**`$(Variable:suffix=ersetzung)`** 

Für `$(Variable)` wird der Wert der Variablen eingesetzt, bei der zweiten Form wortweise am Ende modifiziert. Dabei **rekursives Expandieren**: enthält der Wert wiederum Variablenbenutzungen, wird darauf erneut die Textersetzung angewendet usw.  

Ist eine Variable nicht definiert, wird ihr Wert als leer angenommen. 

## Makefile: Variablen (2)

---

### Sonderfall automatische Variablen:

- vordefinierte Variablen, die bei jeder Regelanwendung einen neuen Wert erhalten, z.B.:

**\$@** Das Ziel, auf das die Regel gerade angewendet wird

**\$<** Die erste Quelle zum aktuellen Ziel

**\$^** Alle Quellen zum aktuellen Ziel

- die automatischen Variablen sind in Musterregeln unentbehrlich, z.B.:

`%: %.c`

`$ (CC) $ (CPPFLAGS) $ (CFLAGS) $ (LDFLAGS) -o $@ $<`

# Makefile: Variablen (3)

---

Vordefinierte Variablen flexibilisieren vordefinierter Musterregeln:

- Kommandos sind in vordefinierten Musterregeln mit Variablen formuliert:

`$ (KOMMANDO)` Wert ist der zu verwendende Befehl

`$ (KOMMANDO)FLAGS` Wert ist zunächst leer 

Beispiele:

`CC=gcc` der C-Compiler (mit Optionen `$ (CFLAGS)` )

`RM=rm -f` der Löschbefehl für Dateien 

- die Werte der Variablen können bei Bedarf überschrieben werden:

in der Aufrufumgebung `export CC="gcc -g"`

im Makefile `CC = gcc -g`

beim Aufruf von make `make "CC=gcc -g"`

*Wert bei Aufruf überschreibt Wert in Makefile überschreibt Wert aus Aufrufumgebung*

# Makefile: Rekursion

---

Große Softwaresysteme bestehen aus vielen Paketen, die Paket für Paket mit **make** erstellt werden müssen.

- die Pakethierarchie wird im Dateisystem als Verzeichnishierarchie abgebildet, mit einem Makefile in jedem Verzeichnis, z.B:

```
hallohallo/   Makefile
├── hallo/    Makefile hallo.c
└── hallo2/   Makefile hallo.c gruss.h gruss.c
```

*Softwaresystem hallohallo  
mit Paketen hallo und hallo2*

- Makefile des Softwaresystems ruft **make** rekursiv für die Pakete auf:

```
# Makefile fuer Softwaresystem hallohallo
PACKAGES=hallo hallo2
.PHONY: all clean
all clean:
    for p in $(PACKAGES); do \
        (cd $$p && $(MAKE) $@); \
    done
```

*Shell-Script als Kommando*



# Makefile: C-Beispiel hallo (1)

**# Makefile**

**# Kommando-Variablen**

CC = gcc

CPPFLAGS = -I.

CFLAGS = -Wall -Wextra -Werror -std=c11 -pedantic

RM = rm -f

*Include-Dateien im aktuellen Verzeichnis suchen*

*die Variablen CC, CPPFLAGS, CFLAGS werden in der vordefinierten C-Übersetzungsregel verwendet*

**# Hilfsvariablen**

TARGET = hallo 

OBJECTS = gruss.o

SOURCES = \$(TARGET).c \$(OBJECTS:.o=.c)

HEADERS = \$(OBJECTS:.o=.h) 

**# Musterregeln**

%.o: %.c

\$(CC) \$(CPPFLAGS) \$(CFLAGS) -c \$< -o \$@


...


*die C-Übersetzungsregel ist vordefiniert und braucht deshalb nicht angegeben zu werden*


# Makefile: C-Beispiel hallo (2)

---

...

```
# Standardziele
.PHONY: all clean ← Pseudoziele
all: $(TARGET)
clean: 
    $(RM) $(TARGET) $(TARGET).o $(OBJECTS) depend
depend: $(SOURCES) $(HEADERS)
    $(CC) $(CPPFLAGS) -MM $(SOURCES) > $@

# Ziele zur Programmerstellung
$(TARGET): $(TARGET).o $(OBJECTS)
    $(CC) $(LDFLAGS) $^ -o $@ 

# Abhaengigkeiten
include depend 
```

# Makefile: Empfehlungen (1)

---

Variablen:

- für jedes in einer Regel verwendete Kommando eine Variable definieren  
*bei Kommandos mit vielen Optionen zusätzliche Variable für Optionen*
- für die Liste der Übersetzungseinheiten / Quelldateien Hilfsvariablen definieren
- in Regeln, wo immer möglich, automatische Variablen verwenden

Regeln:

- wo immer möglich, Musterregeln statt expliziter Regeln verwenden
- immer zumindest die Pseudoziele `a11` und `clean` vorsehen  
*a11 muss das erste Ziel im Makefile sein*  
*clean muss alles beseitigen, was a11 erzeugt*
- Abhängigkeitsregeln möglichst automatisch erzeugen  
*mit einem Ziel `depend` eine gleichnamige Datei erzeugen und per `include` einbinden*

# Makefile: Empfehlungen (2)

---

Vorgehen beim Erstellen: 

- mit der `a11`-Regel beginnen

`a11: Endergebnis`

*Endergebnis ist die zu erstellende Datei  
(bei Bedarf auch mehrere Dateien)*

- für jede bei `a11` als Endergebnis genannte Datei eine Regel erstellen, für jede darin als Zwischenergebnis genannte Datei wiederum eine Regel, usw. bis nur noch Abhängigkeiten von Quelldateien auftreten:

`Endergebnis: Zwischenergebnisse`

`Kommando`

`...`

`Zwischenergebnis: Quelldateien`

`Kommando`

- eine `clean`-Regel erstellen

`clean:`

`$(RM) Endergebnis Zwischenergebnisse`

- mit Variablen und Musterregeln die mehrfache Wiederholung von Dateinamen und Kommandos vermeiden

# Prüfen von Programmen: Fehlersuche (1)

---

Einige wichtige Arten von **Laufzeitfehlern**:

- **vorzeitiges Programmende**

Absturz wegen z.B. Speicherzugriffsfehler oder nicht gefangener Ausnahme

- **Programm terminiert nicht**

Endlosschleife oder Ausführung ist in Wartesituation blockiert ...

- **Speicherüberlauf**

Programm macht den Rechner langsam, weil es übermäßig Speicher belegt  
im Extremfall auch Absturz, weil kein Speicher mehr verfügbar

- **Fehlverhalten**

das Programm tut nicht, was es tun soll, liefert z.B. falsche Ergebnisse

# Prüfen von Programmen: Fehlersuche (2)

---

**Vorgehen** bei der Suche von Laufzeitfehlern:

- Fehler **reproduzieren**

einen Testfall erstellen, bei dem der Fehler auftritt

*oft schwierig bei Programmen mit vielfachen Abhängigkeiten von der Umgebung  
(Benutzer, andere Programme, Zeit, Daten in Dateien oder Datenbanken, Netzwerk, ...)*

- Fehler **isolieren**

mögliche Fehlerursachen schrittweise eingrenzen

*Hypothesen aufstellen und prüfen*

*Programmteile gezielt weglassen oder abändern*

*feststellen, ob ältere oder neuere Programmversionen den Fehler auch zeigen*

*schrittweises Ausführen im Debugger*

# Fehlersuche: Debugger

---

**Debugger** erlauben es, den Programmablauf zu beobachten und zu beeinflussen, ohne den Code dafür aufwändig und fehlerträchtig abzuändern.

Funktionalitäten:

- Programm kontrolliert ausführen  
*Zeile für Zeile, bis Funktionsende, bis zum nächsten Haltepunkt, ...*
- Programm unter bestimmten Bedingungen anhalten lassen  
*unbedingte und bedingte Haltepunkte ("Breakpoints", "Watchpoints")*
- Zustand des angehaltenen Programms untersuchen  
*Aufruf-Stack anzeigen, Speicherinhalte anzeigen, ...*
- Zustand des angehaltenen Programms verändern  
*Speicherinhalte ändern, Anweisungen überspringen, ...*

# Debugger: Nutzen

---

Mit einem Debugger lässt sich meist schnell klären:

- wo ein Programm abstürzt

*Programm mit gleichen Eingaben im Debugger laufen lassen  
oder core-Datei untersuchen*

*(Linux legt bei einem Programmabsturz den gesamten Programmzustand  
in einer Datei `core` ab, einzuschalten mit: `ulimit -c unlimited` ).* 

- wo ein Programm eine Endlosschleife enthält

*Programm im Debugger unterbrechen*

*oder Programm "abschießen" (`kill -6 ...`), um untersuchbare `core-Datei` zu erhalten*

- ob eine Hypothese zur Fehlerursache stimmt

*gezielt Haltepunkte setzen und Programmzustand analysieren*

die Hypothese selbst findet man nur durch **Nachdenken!**



# Debugger: gdb

---

**gdb** –kommandozeilen-orientierter Debugger (wie gcc Teil des GNU-Projekts)

Aufruf:

**gdb Programm [Core-Datei | Prozessnummer]**

- Programm:

Die volle Funktionalität des Debuggers steht nur zur Verfügung, wenn das zu untersuchende Programm mit der gcc-Option **-g** übersetzt wurde.

*es wird dann Information in den Code eingebettet, die dem Debugger den Rückschluss von Adressen auf Variablen und Zeilen im Quellcode erlaubt.*

- Core-Datei:

nur beim nachträglichen Untersuchen eines abgestürzten Programms ("**Post-Mortem-Debugging**").

- Prozessnummer:

zum nachträglichen Ankoppeln des Debuggers an ein laufendes Programm

# Speicherfehler suchen: valgrind

---

**valgrind** – ein Speicherdebugger für x86-Linux 

Aufruf:

**valgrind [Optionen] Programm [Argumente]**

- Funktionsweise:  
interpretiert x86-Maschinencode (virtueller Prozessor) und führt dabei Buch über die Speichernutzung des Programms.  
*Das Programm läuft dadurch langsamer und braucht mehr Speicher.*
- Fehlererkennung:  
Lesezugriff auf nicht initialisierten Speicher bei Verzweigungen  
Lese- oder Schreibzugriff auf nicht reservierten Heapspeicher  
Feldgrenzen-Überschreitung für separat auf dem Heap allokierte Felder  
Speicherlecks (**malloc/calloc** ohne zugehöriges **free**)  
doppeltes Freigeben von reserviertem Speicher (mehrfaches **free**)

# Werkzeuge: Index

---

`$<` 8-21  
`$@` 8-21  
`$^` 8-21  
`.PHONY` 8-19  
Abhängigkeitsregel 8-15,8-17  
Absturz 8-28,8-31  
`all` 8-18  
`astyle` 8-3  
automatische Variable 8-21  
`bash` 8-10  
Breakpoint 8-30  
`clean` 8-18  
Debugger 8-30,8-31  
`diff` 8-4  
Endlosschleife 8-28,8-31  
explizite Regel 8-15,8-16  
Fehlverhalten 8-28  
`find` 8-5  
`gcc` 8-5  
`gdb` 8-32  
`grep` 8-4  
Haltepunkt 8-31  
Kommandoprozedur 8-9 bis 8-11  
`make` 8-12 bis 8-14  
Makefile 8-13,8-15 bis 8-27  
Musterregel 8-15,8-19  
Programmierwerkzeuge 8-1  
Pseudoziel 8-18  
Post-Mortem-Debugging 8-32  
`sh` 8-10  
shell-Script 8-10,8-11  
Speicherdebugger 8-34  
Speicherüberlauf 8-29  
`valgrind` 8-33  
Variable 8-21 bis 8-23  
Watchpoint 8-30