

# Systemprogrammierung

## Teil 5: 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 ANSI-C: Präprozessor, Compiler, Linker*

#### Prüfen von Programmen

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

- **Ermöglichung**

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

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. *Formatierungsmängel beseitigen mit Werkzeug astyle*).

## Bearbeiten von Quellcode: Formatierung

---

**astyle** – ein "Beautifier" 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 -rl "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 des Quellcodes:

```
-W -Wall -ansi -pedantic
```

*vor potenziellen  
Fehlern warnen*

*Einhaltung des ANSI-C-Sprachstandards überwachen*

# 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

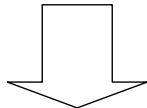
---

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  
und im Fehlerfall unnötige Folgefehler produzieren.*

**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"      # Variable mit Initialisierung
    echo $compile_command            # Wert der Variablen ausgeben
    eval $compile_command            # Wert der Variablen als Kommando ausführen
    if [ $? -ne 0 ] ; then          # Rückgabewert des Kommandos prüfen
        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
```

# Build-Werkzeug: 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

## 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 Objektdateien)*

*Kommando  
(erzeugt hallo aus  
zwei Objektdateien)*

*Tabulator vor dem Kommando nicht vergessen*

- Aufruf zum Erstellen bzw. Aktualisieren des Programms:

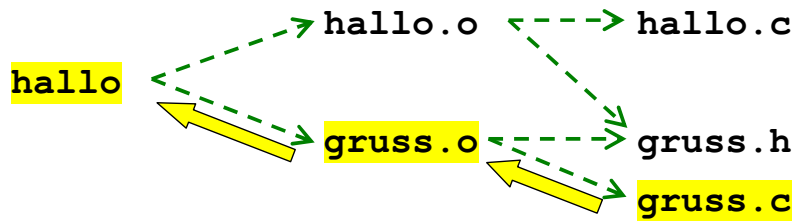
```
make -f Makefile hallo
```

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

## make: Beispiel (2)

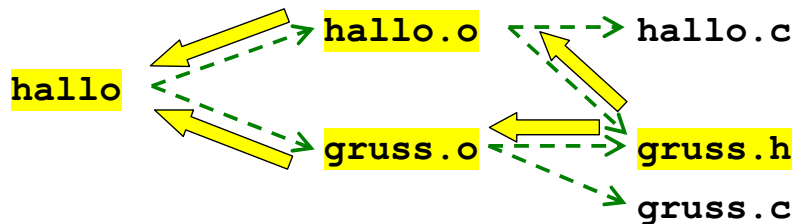
Abhängigkeiten (→) steuern das **inkrementelle Erstellen** (←):

- Aufruf nach Änderung von **gruss.c**



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

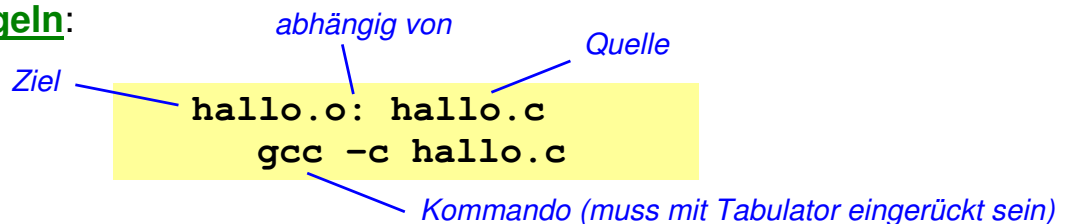
- Aufruf nach Änderung von **gruss.h**



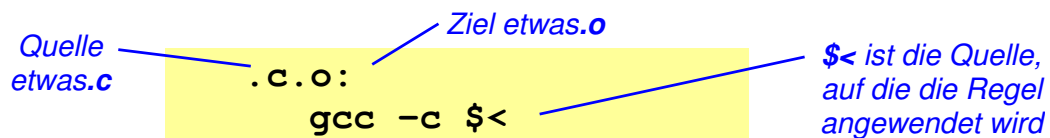
*alles wird  
neu erstellt,  
weil abhängig  
von gruss.h*

## Makefile: Regeln

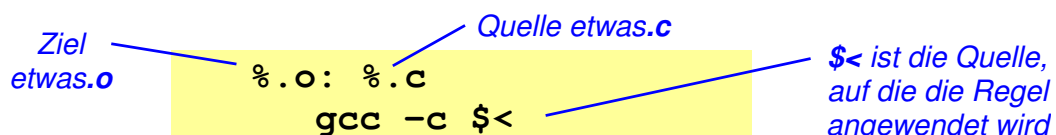
- explizite Regeln:



- implizite Regeln: beziehen sich auf Dateiendungen



- Musterregeln: explizite Regeln mit Platzhalter % für beliebige Zeichenfolgen





## Makefile: explizite Regeln (1)

---

```
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 kein (→ *Sonderformen*) oder ein Kommando pro Regel,  
auch komplizierte Kommandos in Shell-Skript-Syntax möglich.  
Liefert ein Kommando einen Fehlerstatus, beendet sich make automatisch.

## Makefile: explizite Regeln (2)

---

Sonderform **Abhängigkeitsregel**:

- 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 implizite oder Musterregeln, die die Kommandos festlegen, z.B.:

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

- Abhängigkeitsregeln kann gcc automatisch aus den C-Quellen erzeugen, indem er die `#include`-Anweisungen auswertet:

```
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: explizite Regeln (3)

---

Sonderform mit **Pseudoziel**:

- 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
```

*Pseudoziele **all**, **clean**, **install**, **uninstall***

*haben sich als Quasistandard eingebürgert*

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

```
all: hallo
```

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

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

```
clean:
```

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

## Makefile: implizite Regeln (1)

---

```
.Quellendung.Zielendung:  
<TAB>Kommando
```

Anwendung auf Dateitypen, wobei die Dateiendung den Typ bestimmt:

- Alle Endungen müssen beim eingebauten Ziel **.SUFFIXES** genannt sein:

```
.SUFFIXES: .c .o
```

- **.Quellendung** legt den Dateityp der Quelle fest, den die Regel benötigt.

Der Name der Quelle **Name.Quellendung** muss mit dem Namen der Zielfeile übereinstimmen, für die die Regel ausgeführt wird.

- **.Zielendung** legt den Dateityp fest, auf den die Regel anwendbar ist.

Die Regel wird für beliebige Zielfeile **Name.Zielendung** ausgeführt, falls es für die Datei keine explizite Regel mit Kommando gibt.

- **Kommando** ist ein Befehl, der die Zielfeile aus der Quelldatei erzeugt.

## Makefile: implizite Regeln (2)

make hat für die wichtigsten Dateitypen vordefinierte implizite Regeln, z.B.:

- Übersetzen und binden eines C-Programms mit nur einer Quelldatei:

`.c:`  Zielendung ist leer (ausführbare Programme unter Linux ohne Endung)

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

*make hallo erzeugt aus hallo.c das ausführbare Programm hallo*

*automatische  
Variablen für  
Ziel und Quelle*

- Übersetzen einer C-Quelle:

`.c.o:`

`$(CC) $(CFLAGS) $(CPPFLAGS) -c $<`

*make hallo.o erzeugt aus hallo.c die Objektdaten hallo.o*

- Kommandos in vordefinierten Regeln sind mit **Variablen** definiert, um sie leicht an unterschiedliche Plattformen anpassen zu können:

`CC=gcc`

`CPPFLAGS=`  Präprozessor-Optionen (hier leer)

`CFLAGS=`  Compiler-Optionen (hier leer)

`LDFLAGS=`  Linker-Optionen (hier leer)

## Makefile: Musterregeln

**Zielmuster: Quellmuster**  
**<TAB>Kommando**

Ziel- und Quellmuster enthalten **%** als Platzhalter für beliebige Zeichenkette:

- Übersetzen und binden eines C-Programms mit nur einer Quelldatei:

`%: %.c`

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

*z.B. ausführbares Programm hallo aus der C-Quelle hallo.c erzeugen*

- Übersetzen einer C-Quelle:

`%.o: %.c`

`$(CC) $(CFLAGS) $(CPPFLAGS) -c $<`

*z.B. Objektdaten hallo.o aus der C-Quelle hallo.c erzeugen*

- Musterregeln wurden als flexiblerer Ersatz für implizite Regeln eingeführt

## Makefile: Variablen (1)

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

- Variablendefinition:

**Variable = Wert** oder mehrzeilig

Variable üblicherweise in Großbuchstaben  
Wert ist eine beliebige Zeichenkette

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

*Zeilenwechsel  
müssen mit \  
maskiert werden*

- Variablenbenutzung:

**\$(Variable)**

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

Textersetzung von **\$(Variable)** durch den Wert der Variablen  
bzw. durch den wortweise am Ende modifizierten Wert der Variablen

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

**Rekursives Expandieren**: enthält der Wert wiederum Variablenbenutzungen,  
wird darauf erneut die Textersetzung angewendet, usw.

## 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 impliziten Regeln und Musterregeln unentbehrlich, z.B.:

**.c:**

**\$(CC) \$(CPPFLAGS) \$(CFLAGS) \$(LDFLAGS) -o \$@ \$<**

bzw.

**%: %.c**

**\$(CC) \$(CPPFLAGS) \$(CFLAGS) \$(LDFLAGS) -o \$@ \$<**

## Makefile: Variablen (3)

Vordefinierte Variablen als Parametrierung eingebauter impliziter Regeln:

- Kommandos sind in impliziten Regeln indirekt mit Variablen formuliert:

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

`$(KOMMANDOFLAGS)`              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 nach 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.

- Pakethierarchie wird im Dateisystem als Verzeichnishierarchie abgebildet,  
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*

## Makefile: C-Beispiel hallo (1)

```
# Makefile
# Kommando-Variablen
CC = gcc
CFLAGS = -W -Wall -ansi -pedantic
CPPFLAGS = -I.
RM = rm -f

# Hilfsvariablen
TARGET = hallo
OBJECTS = gruss.o
SOURCES = $(TARGET).c $(OBJECTS:.o=.c)
HEADERS = $(OBJECTS:.o=.h)

# Musterregeln
%.o: %.c
    $(CC) $(CPPFLAGS) $(CFLAGS) -c $< -o $@
...

```

*Variablen für die vordefinierten C-Übersetzungsregeln*

*Include-Dateien im aktuellen Verzeichnis suchen*

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

## Makefile: C-Beispiel hallo (2)

```
...
# Standardziele
.PHONY: all clean
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 $@

# Abhängigkeiten
include depend

```

*Pseudoziele*

## Makefile: Empfehlungen (1)

---

Variablen:

- für jedes in einer Regel verwendete Kommando eine Variable definieren  
*bei komplexen Kommandos 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 **all** und **clean** vorsehen  
*all muss das erste Ziel im Makefile sein*  
*clean muss alles beseitigen, was all 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 **all**-Regel beginnen  
**all: *Endergebnis*** *Endergebnis ist die zu erstellende Datei (bei Bedarf auch mehrere Dateien)*
- für jede bei **all** 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**:

- **Absturz**  
unerwartetes Programmende, z.B. wegen Speicherzugriffsfehler
- **Endlosschleife**  
das Programm scheint zu "hängen", aber es läuft und läuft und läuft ...
- **Speicherüberlauf**  
der ganze Rechner wird langsam,  
weil das Programm sämtlichen Speicher belegt hat
- **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 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: ddd

---

**ddd** – der GNU Data-Display-Debgger, eine graphische Benutzeroberfläche für den kommandozeilen-orientierten GNU-Debugger **gdb**.

Aufruf:

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

- Programm:

Die volle Funktionalität des Debuggers steht nur zur Verfügung, wenn das 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

---

<b>\$&lt;</b>	5-24	<b>gcc</b>	5-6
<b>\$@</b>	5-24	<b>grep</b>	5-5
<b>\$^</b>	5-24	Haltepunkt	5-33
<b>.PHONY</b>	5-19	implizite Regel	5-16
Abhängigkeitsregel	5-18	Kommandoprozedur	5-10 bis 5-12
Absturz	5-31,5-34	<b>make</b>	5-13 bis 5-15
<b>all</b>	5-19	Makefile	5-13,5-16
<b>astyle</b>	5-4	Musterregel	5-16
automatische Variable	5-24	Programmierwerkzeuge	5-1
<b>bash</b>	5-11	Pseudoziel	5-19
Breakpoint	5-33	Post-Mortem-Debugging	5-35
<b>clean</b>	5-19	<b>sh</b>	5-11
<b>ddd</b>	5-35	shell-Script	5-11,5-12
Debugger	5-33	Speicherdebugger	5-36
<b>diff</b>	5-5	Speicherüberlauf	5-31
Endlosschleife	5-31,5-34	<b>valgrind</b>	5-36
explizite Regel	5-16 bis 5-19	Variable	5-23 bis 5-25
Fehlverhalten	5-31	Watchpoint	5-33
<b>find</b>	5-5		