Systemprogrammierung

Teil 7: Werkzeuge

Programmerstellung, Fehlersuche

Werkzeuge: Einsatzgebiete

Erstellen von Programmen

- <u>Bearbeiten</u> von Quellcode
 Schreiben von neuem Quellcode
 Ändern von vorhandenem Quellcode
- <u>Transformieren</u> 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. pluma)
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

• 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 ...][-Idir ...] Präprozessor steuern

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

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

[-Olevel] Code-Optimierung steuern

[-Ldir ...] [-1name ...] 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 Warnungen als Einhaltung des Sprachstandards
Fehlern warnen Fehler werten überwachen (hier ISO C11)
```

7-5

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
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 mehren Übersetzungseinheiten: 🗾

• jede Übersetzungseinheit getrennt <u>übersetzen</u>:

```
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.

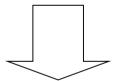
Transformieren von Quellcode: Probleme

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 <u>alle</u> 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 <u>nur die laut Bauplan erforderlichen</u> Kommandos ausgeführt.

Kommandoprozedur: Linux Shell-Script (1)

• eine <u>Linux-Shell</u> 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 <u>Shell-Script</u> 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

7-10

Kommandoprozedur: Linux Shell-Script (2) 5

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

```
#!/bin/sh
for s in hallo.c gruss.c; do
                                    # Variable mit Initialisierung 📮
  compile command="qcc -c $s"
                                    # Wert der Variablen ausgeben 📃
  echo $compile command
  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: 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:

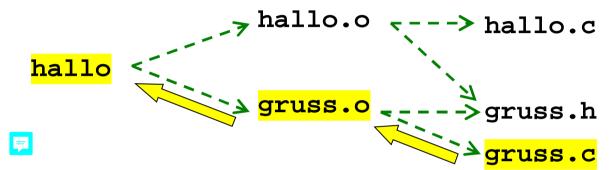
• Aufruf zum Erstellen bzw. Aktualisieren des Programms:

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

GNU make: Beispiel (2)

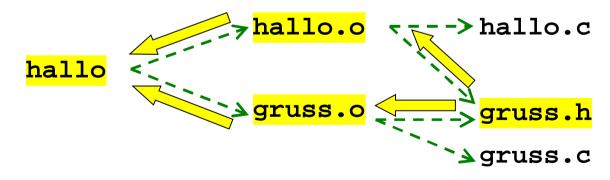
Abhängigkeiten (-->) steuern das inkrementelle Erstellen (<---):

• Aufruf nach Änderung von gruss.c



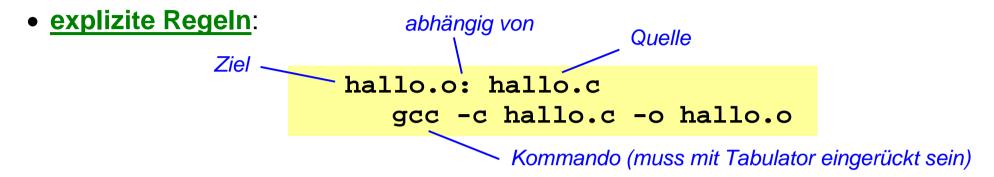
hallo.o wird <u>nicht</u> neu erstellt, weil unabhängig von gruss.c

• Aufruf nach Änderung von gruss.h



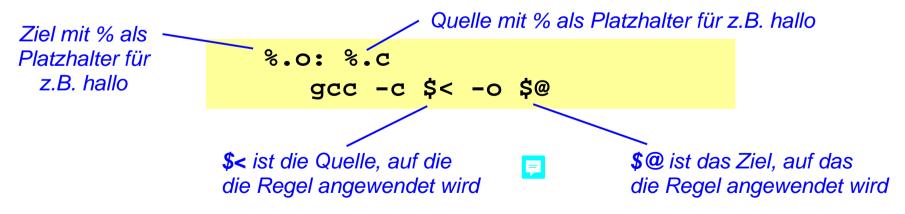
alles wird neu erstellt, weil abhängig von gruss.h

Makefile: Regeln



• Abhängigkeitsregeln sind explizite Regeln ohne Kommando:

• Musterregeln liefern das Kommando zu gleichartigen Abhängigkeitsregeln:



7-15

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.:

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++):

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 <u>Pseudoziel</u> 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 .рному-Regel:

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

die 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:

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 $@ $
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)

Prof. Dr. H. Drachenfels

Systemprogrammierung

7-19

Hochschule Konstanz

Makefile: Variablen (1)

Mit <u>Variablen</u> können häufig wiederkehrende Bestandteile von Makefiles zusammengefasst und parametrisiert werden.

Variablendefinition:

Variable = Wert oder mehrzeilig



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 <u>rekursives Expandieren</u>: 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 <u>automatische Variablen</u>:

- 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) -0 $ < $ <
```

Makefile: Variablen (3)

Vordefinierte Variablen flexibilisieren vordefinierter Musterregeln:

• Kommandos sind in vordefinierten Musterregeln 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 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 Softwaresystem hallohallo
hallo/ Makefile hallo.c mit Paketen hallo und hallo2
hallo2/ Makefile hallo.c gruss.h gruss.c
```

• Makefile des Softwaresystems ruft make rekursiv für die Pakete auf:

7-23

Makefile: C-Beispiel hallo (1)

```
# Makefile
# Kommando-Variablen
                           Include-Dateien im aktuellen Verzeichnis suchen
CC = qcc
CPPFLAGS = -I.
CFLAGS = -Wall -Wextra -Werror -std=c11 -pedantic
RM = rm - f
                            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)
                                   die C-Übersetzungsregel ist vordefiniert und
# Musterregeln
                                   braucht deshalb nicht angegeben zu werden
%.O: %.C
   $(CC) $(CPPFLAGS) $(CFLAGS) -c $< -o $@
```

Systemprogrammierung

Makefile: C-Beispiel hallo (2)

```
Pseudoziele
# 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 $@
# 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 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 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 oder nicht gefangener Ausnahme

• 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 oder neuere Programmversionen den Fehler auch zeigen

schrittweises Ausführen im Debugger

Fehlersuche: Debugger

<u>Debugger</u> 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 ("<u>Breakpoints</u>", "<u>Watchpoints</u>")
- 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 <u>D</u>ata-<u>D</u>isplay-<u>D</u>ebugger, 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 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:

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

\$< 7-21 gcc 7-5 \$@ 7-21 grep 7-4 \$^ 7-21 Haltepunkt 7-31 .PHONY 7-19 Kommandoprozedur 7-9 bis 7-11 Abhängigkeitsregel 7-15,7-17 make 7-12 bis 7-14 Absturz 7-28,7-31 Makefile 7-13,7-15 bis 7-27 all 7-18 Musterregel 7-15,7-19 astyle 7-3 Programmierwerkzeuge 7-1 automatische Variable 7-21 Pseudoziel 7-18 bash 7-10 Post-Mortem-Debugging 7-32 Breakpoint 7-30 sh 7-10 clean 7-18 shell-Script 7-10,7-11 Speicherdebugger 7-34 **ddd** 7-32 Speicherüberlauf 7-29 Debugger 7-30,7-31 valgrind 7-33 diff 7-4 Variable 7-21 bis 7-23 Endlosschleife 7-28,7-31 Watchpoint 7-30 explizite Regel 7-15,7-16 Fehlverhalten 7-28 find 7-5