

# Entwicklungswerkzeuge

In diesem Teil der Veranstaltung wird in die Werkzeuge zur Entwicklung von Software eingeführt, die hauptsächlich im Umfeld freier Software verwendet werden.

## Inhalt

|   |                    |
|---|--------------------|
| <a href="#">1. GNU Software.....</a>                          | <a href="#">3</a>  |
| <a href="#">2. make.....</a>                                  | <a href="#">4</a>  |
| <a href="#">2.1. Grundlagen.....</a>                          | <a href="#">5</a>  |
| <a href="#">2.2. Makros.....</a>                              | <a href="#">13</a> |
| <a href="#">2.3. Aufrufoptionen von make.....</a>             | <a href="#">15</a> |
| <a href="#">3. Übersetzen mit gcc.....</a>                    | <a href="#">16</a> |
| <a href="#">3.1. Übersetzung einfacher Programme.....</a>     | <a href="#">16</a> |
| <a href="#">3.2. Programme mit mehreren Quelldateien.....</a> | <a href="#">18</a> |
| <a href="#">3.3. Bibliotheken erzeugen .....</a>              | <a href="#">20</a> |
| <a href="#">3.4. Header Dateien.....</a>                      | <a href="#">21</a> |

|   |    |
|---|----|
| 4. GNU Build System .....                           | 22 |
| 4.1. Erstes Beispiel mit autoconf und automake..... | 23 |
| 5. Werkzeuge zur Programmanalyse.....               | 29 |
| 5.1. Laufzeitanalyse.....                           | 29 |
| 5.1.1. time.....                                    | 30 |
| 5.1.2. Gprof.....                                   | 31 |
| 5.1.3. gcov.....                                    | 39 |
| 5.2. Statische Programmanalyse mit splint.....      | 42 |
| 6. rcs.....   | 43 |
| 6.1. Prinzip der Versionsführung.....               | 43 |
| 6.2. Beispielsitzung.....                           | 46 |

## 1. GNU Software

Freie Software ist meist als Quellcode-Distribution verfügbar. Zur Installation sind folgende Schritte erforderlich:

- anpassen von Make-Files
- anpassen von Header-Dateien
- übersetzen der angepassten Quellen
- Installation in Zielumgebung

Diese Schritte sind fehleranfällig. Deshalb wurde für GNU-Projekte ein Mechanismus entwickelt, der die o.a. Schritte automatisiert:

- `./configure`: Konfiguration der Quellen
- `make`: übersetzen der Quellen
- `make install`: Installation

Die einzelnen Werkzeuge zur o.a. automatisierten Installation werden hier beschrieben.

## 2.make

Große **Programmpakete** sollen **modular** aufgebaut werden. Durch diese Modularisierung entstehen viele **Dateien**, die funktional **voneinander abhängen**. Diese Abhängigkeiten bestimmen die **Übersetzungsreihenfolge** der einzelnen C-Quellen. Ab einer bestimmten Paketgröße (Anzahl der Dateien) wird der Prozess der Erzeugung eines ausführbaren Objektes nicht mehr handhabbar.

Unix stellt dazu ein Werkzeug zur Verfügung: **make**.

Make kann überall dort eingesetzt werden, wo bestimmte Aktionen notwendig sind, sobald eine konkrete Datei verändert wurde, z.B.

- bei allen Programmierprojekten
- bei Backups: "Wurde(n) diese Datei(en) modifiziert, erzeuge die Backup-Datei neu."
- bei der Systemverwaltung: "Wurde diese Konfigurationsdatei verändert, dann generiere eine neue Datenbasis." (dieses Prinzip wird u.a. für die NIS-Datenbasis angewandt.)
- bei LaTeX-Dokumenten: "Erzeuge die Postscript-Datei neu, falls eine der Quelldateien geändert wurde."

Wird an einer Datei eine **Änderung** vorgenommen, so kann eine lauffähige Version erzeugt werden, indem man „make“ veranlasst, die **Abhängigkeiten** zu **prüfen** und die betroffenen Quelldateien (nur diese) neu zu **übersetzen**.

Die **Abhängigkeiten** werden in einem File (**Makefile**) abgelegt; „make“ liest diese Datei und überprüft, ob es Dateien gibt, die von „**jüngeren**“ Dateien **abhängen**. In diesem Fall müssen diese abhängigen „**älteren**“ Dateien **neu übersetzt** werden.

## 2.1. Grundlagen

Im einfachsten Fall wird »make« ohne Argumente gestartet:

```
$ make
```

Make sucht jetzt im aktuellen Verzeichnis nach einer Datei mit dem Namen

1. »GNUmakefile«,
2. »makefile« oder
3. »Makefile«.

Make sucht in der o.a. Reihenfolge und verwendet die **erste** gefundene Steuerdatei (und nur diese!). Findet make kein Makefile, so ist die Antwort:

```
$ make
make: *** No targets. Stop.
$
```

In einem Makefile sind die Abhängigkeiten zu den einzelnen Zielen (Targets) beschrieben. Per Voreinstellung beginnt »make« mit der Abarbeitung des ersten Zieles aus der Steuerdatei und endet, sobald alle Aktionen zu **diesem** Ziel erledigt sind. Makefiles bestehen aus **Regeln** (»rules«).

Jede Regel besitzt folgendes Format:

**Ziel:**   **Abhängigkeiten**

**Kommando1**

**Kommando2**

    ...

- |                |   |
|----------------|---|
| Ziel           | Ziel ("target") ist normalerweise der Name des Programm(moduls), das durch die nachfolgenden Kommandos generiert wird. Ziel kann aber auch der Name einer Aktion sein.  |
| Abhängigkeiten | Hier stehen meist die Namen der Dateien und Ziele, von denen dieses Ziel abhängt.   |
| Kommando n     | Hier stehen die Aktionen, die im Falle einer erfüllten Abhängigkeit auszuführen sind. Jede Aktion muss auf einer eigenen Zeile stehen und durch einen Tabulator eingerückt sein ( <b>keine Leerzeichen</b> ). Sehr lange Zeilen können umgebrochen werden, indem am Ende jeder Zeile - bis auf die letzte - ein Backslash »\« angefügt wird (auch hier gilt: keine weiteren Zeichen nach dem Backslash!). |

Ein einfaches Makefile ohne Abhängigkeiten ist:

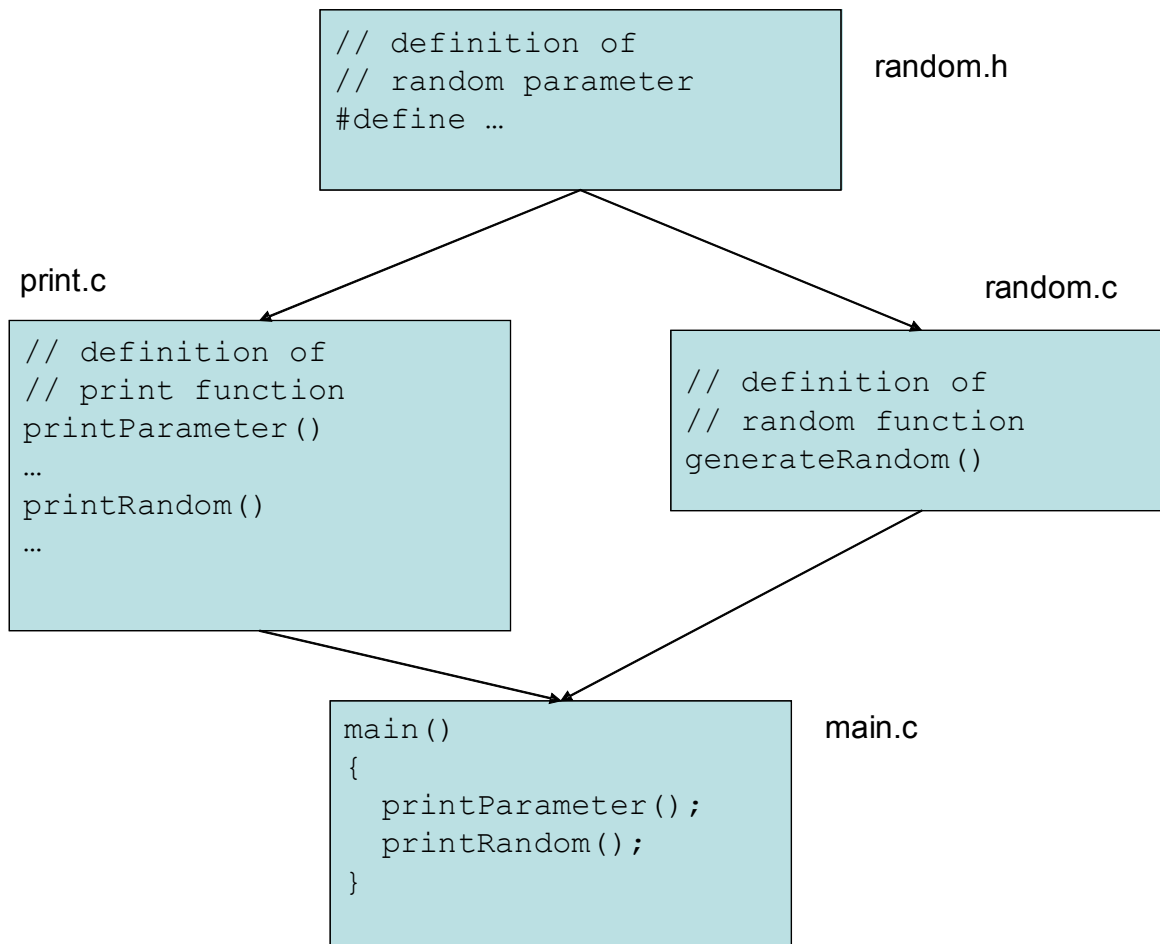
```
$ cat Makefile
clean:
    rm -f *.tmp *.log *.tar
test:
    date;ls
$
```

Dabei werden durch „`make clean`“ alle Dateien mit Endung `tmp`, `log` und `tar` gelöscht.

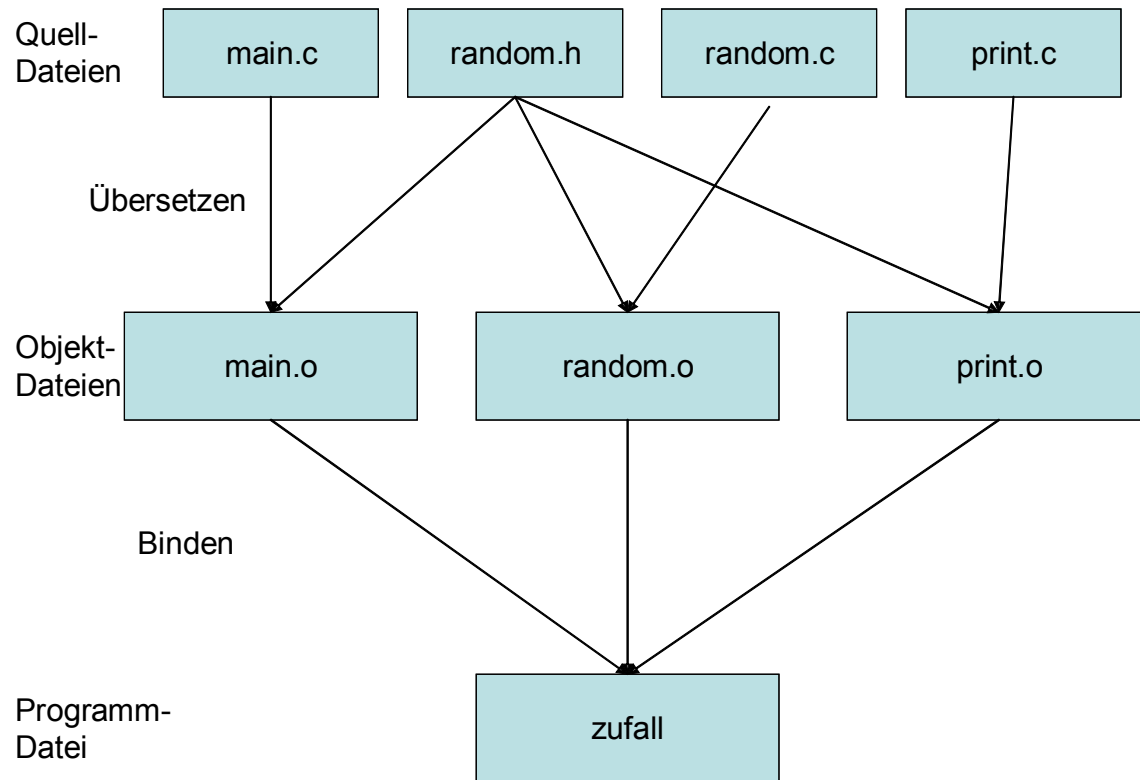
Verdeutlicht wird die Arbeitsweise von `make` zur Programmentwicklung durch folgendes Szenario.

Ein Programm „zufall“ soll eine Folge von 10 Zufallszahlen auf den Bildschirm schreiben. Verwendet werden soll der Algorithmus von Knuth.

Das Programm soll folgende modulare Struktur aufweisen:



Neben den „logischen“ Abhängigkeiten, gibt es bei C-Programmen die Abhängigkeiten:



Das Makefile, mit dem man diese Anwendung übersetzen und binden kann hat folgendes Aussehen:

```
$ cat makefileC
zufall: zufall.o print.o random.o
        gcc zufall.o print.o random.o -o zufall
zufall.o: main.c
```

```
        gcc -c main.c -o zufall.o
print.o: print.c
        gcc -c print.c
random.o: random.c
        gcc -c random.c
clean:
        rm *.o
$
```

Die einzelnen zum „Programmpaket“ gehörenden Dateien sind:

```
$ cat random.h
#define R 1000001
#define a 100001
#define m 1717
#define c 3
$
```

```
$ cat main.c
extern void printParameter();
extern void printRandom();
extern float generateRandom();
main() {
    int i;
    printParameter();
    for (i=1;i<=10;i++)
        printRandom(generateRandom());
}
$
```

```
$ cat random.c
#include "random.h"
float generateRandom() {
    static long int r=R;
    r = (a*r)%m;
    return r/(float)m;
}
$
```

```
$ cat print.c
#include "random.h"
#include <stdio.h>
void printParameter() {
    printf("a=%ld\n", a);
    printf("c=%ld\n", c);
    printf("m=%ld\n", m);
}

void printRandom(float i) {
    printf("%f\n", i);
}
$
```

Das o.a. Projekt kann nun mit dem Makefile wie folgt übersetzt werden:

```
$ make -f makefileC1
```

->am Rechner: was passiert, wenn random.h verändert wird?

Sind mehrere Aktionen erforderlich, um ein Ziel zu erreichen, so werden die dazugehörigen Kommandos untereinander geschrieben. Wenn bei einem der Kommandos ein Fehler auftritt, so wird die Ausführung des Makefiles abgebrochen. Dies kann durch die Option „-i“ von make unterbunden werden.

Normalerweise werden die Kommandos am Bildschirm angezeigt. Ein vorangestellter Klammerschließen (@) unterdrückt die Bildschirmausgabe.

## **2.2.Makros**

Ähnlich der Shell kennt make Makros bzw. Variablen:

Ein Makro ist durch Zuweisung von Werten definiert; seinen Wert erhält man durch vorangestelltes Dollar-Zeichen. Der Name muss in runden Klammern eingeschlossen werden, wenn er aus mehr als einem Buchstaben besteht.

Damit sieht unser Makefile schon etwas professioneller aus:

```
$ cat makefileC2
OBJECTS=zufall.o print.o random.o
LIBS=-lm
CC=gcc

zufall: $(OBJECTS)
    $(CC) $(OBJECTS) -o zufall
zufall.o: main.c
    $(CC) -c main.c -o zufall.o
print.o: print.c
    $(CC) -c print.c
random.o:
    $(CC) -c random.c
clean:
    @rm *.o

$
```

Neben benutzerdefinierten Makros existieren vordefinierte Makros:

- `$$` der Zielname, also die Datei, die durch make erzeugt werden soll.
- `$$?` Zeichenkette aller Namen, die „jünger“ sind, als das zu erzeugende Ziel
- `$$<` der Name der Datei, die die Aktion ausgelöst hat
- `$$*` der Präfix der Datei (also x bei x.o)

Daneben gib es Makros, die zwar definiert sind, aber keine initialen Werte haben, z.B CFLAGS für Optionen des C-Compilers. Soll z.B. der C-Compiler immer mit der Option -O (optimiere) aufgerufen werden, so würde man im Makefile CFLAGS=-O am Anfang definieren.

Eine Menge von vordefinierten Regeln (systemabhängig) existiert, z.B. dass aus einer .c Datei durch Anwendung von \$(CC) eine .o Datei wird. Deshalb konnte die Abhängigkeit im letzten Makefile bei random.o weggelassen werden.

### **2.3.Aufrufoptionen von make**

Die Syntax des Aufrufs ist:

```
make [Optionen] [Makro-Definitionen] [Ziele]
```

Optionen:

- f file die Datei file wird als Makefile verwendet
- p Ausgabe der Liste der vordefinierten Makros, Ziele und Regeln
- i Fehler bei Aktionen führen nicht zum Abbruch
- n Kommandos der Aktionen nicht ausführen, nur anzeigen
- s Silent Mode: keine Ausgabe der Kommandos

## 3. Übersetzen mit gcc

### 3.1. Übersetzung einfacher Programme

Ein einfaches Programm wird wie folgt übersetzt.

```
$ cat hello.c
#include <stdio.h>
int main () {
    printf ("Hello world!\n");
}
$
$ gcc hello.c
$
$ ls -l
total 56
-rwx-----  1 as  as  11532 Nov 21 12:41 a.out
-rw-----  1 as  as    64 Nov 21 12:05 hello.c
$
$ ./a.out
Hello world!
```

Das übersetzte Programm heißt a.out und wurde ausgeführt.

## Der C-Compiler kennt mehrere Optionen, z.B. -o

```
$ gcc hello.c -o hello
$ ls -l
total 56
-rwx-----  1 as  as  11532 Nov 21 12:41 a.out
-rwx-----  1 as  as  11532 Nov 21 12:41 hello
-rw-----  1 as  as    64 Nov 21 12:05 hello.c
$
```

### Weitere wichtige Optionen:

- -g  
**Debugg** Informationen werden in die ausführbare Datei geschrieben. Dies ist erforderlich, damit ein Debugger (z.B. gdb) arbeiten kann.
- -O, -O2, -O3  
**Optimierungsoptionen**; normalerweise werden diese Optionen während der Programm-entwicklung nicht verwendet. Man sollte sie dennoch am Ende der Entwicklung einschalten zum Testen: wenn jetzt noch **Programmabbrüche** auftauchen, ist es ein Hinweis auf **Speicherfehler**.
- -WALL  
Warnungen bei schlechtem Programmierstiel werden ausgegeben.

## 3.2. Programme mit mehreren Quelldateien

Soll ein Programm aus mehreren Quelldateien aufgebaut werden, kann man alle beim Aufruf von gcc angeben:

```
$ gcc main.c print.c random.c
$
```

Oder alternativ jede Quelldatei übersetzen und am Ende alles zusammenbinden:

```
$ gcc -c random.c
$ gcc -c print.c
$ gcc -c main.c
$ ls -l
total 96
-rw----- 1 as  as   215 Nov 21 13:05 main.c
-rw----- 1 as  as  1004 Nov 21 13:14 main.o
-rw----- 1 as  as   209 Nov 21 13:07 print.c
-rw----- 1 as  as  1108 Nov 21 13:14 print.o
-rw----- 1 as  as   124 Nov 21 13:06 random.c
-rw----- 1 as  as    62 Nov 21 13:05 random.h
-rw----- 1 as  as   936 Nov 21 13:13 random.o
$ gcc print.o random.o main.c
```

Werden nun Änderungen an einer Quelldatei gemacht, muss nur diese Quelldatei neu übersetzt werden und anschließend das Programm gebunden werden.

```
$ gcc -c print.c  
gcc print.o random.o main.o
```

Symbolische Referenzen in Objektdateien kann man durch das Kommando „nm“ auflisten lassen:

```
$ nm main.o  
          U _generateRandom  
00000000 T _main  
          U _printParameter  
          U _printRandom  
          U dyld_stub_binding_helper  
$
```

- Die 1. Spalte ist die symbolische Adresse im Objektfile,
- die 2. der Typ (T=Funktionsdefinition, t=static Funktion, D=globale Variable, R=Konstante, U=Symbol (nicht in File definiert))
- die 3. der Name der symbolischen Referenz.

Der **Compiler** erzeugt **Objektdateien**, der **Linker** setzt alle **Objekte** zu einer **ausführbaren Datei zusammen**, indem er die symbolischen Referenzen je Objektdatei anpasst, so dass sie relativ zu der ausführbaren Datei sind.

### 3.3. Bibliotheken erzeugen

Eine **Menge** von zusammengehörenden **Objektdateien** kann man als **Bibliothek** zusammenfassen.

```
$ ar cru libmyrand.a random.o print.o
$ ls -l *.a
-rw----- 1 as as 2K Nov 21 13:45 libmyrand.a
$
```

Auf vielen Unix-Systemen muss nach der Erzeugung einer Bibliothek, das Kommando **ranlib** ausgeführt werden, wodurch in der Bibliothek eine Symboltabelle zugefügt wird (historisch entstanden).

```
$ ranlib libmyrand.a
$
```

Nun kann das Programm main.c auch übersetzt werden unter Verwendung der Bibliothek libmyrand.a

```
$ gcc main.c libmyrand.a -o zufall
$
```

Soll die Bibliothek **anderen Projekten** verfügbar gemacht werden, kann sie in das Verzeichnis **„/usr/local/lib“** kopiert werden oder in ein eigens lib-Verzeichnis. Dann kann der Aufruf von gcc wie folgt erfolgen:

```
$ gcc main.c -lmyrand -o zufall
$ gcc main.c -L $HOME/as/lib -lmyrand
```

### 3.4.Header Dateien

Normalerweise besteht ein Quellprogramm aus einer Menge von „\*.c“ und „.h“ Dateien. Die Headerdateien deklarieren die Ressourcen, die in Bibliotheken definiert sind.

Die **Headerdateien** für eigenimplementierte Bibliotheken kann man in das Standardverzeichnis „**/usr/local/include**“ kopieren. Dann kann man die Deklaration als

```
#include <myHeader.h>
```

angeben. Alternativ kann man ein eigenes Verzeichnis anlegen, die Datei dorthin kopieren und gcc wie folgt aufrufen:

```
$ gcc -c -I$HOME/include quelle.c
```

## 4. GNU Build System

Das GNU Build System verfolgt zwei Ziele:

- Vereinfachung der Entwicklung **portabler** Programme (-> configure)
- Einfache Verteilung von Software als Quellcode (-> automatische Erstellung von Makefiles)

Dazu stehen mehrere Werkzeuge zur Verfügung:

- autoconf  
**erzeugt** ein Konfigurationsscript (**configure**), das die Installationsumgebung auf Portabilität prüft und spezifische Dateien erzeugt, die zur Installation gebraucht werden.
- automake  
**erzeugt** Makefile-Schablonen (**Makefile.in**), die von autoconf verwendet werden. Basis ist eine Beschreibung in „**Makefile.am**“.
- libtool  
ermöglicht die Erzeugung von portablen gemeinsam verwendbaren Bibliotheken (**shared libraries**)

## 4.1. Erstes Beispiel mit autoconf und automake

Wir werden das hello-world Programm mit autoconf und automake erzeugen und eine Distribution zusammenstellen.

Ausgangspunkt ist lediglich das C-Programm:

```
$ cat hello.c
#include <stdio.h>
int main () {
    printf ("Hello world!\n");
}
$
```

Zunächst werden die Dateien Makefile.am und configure.in angelegt:

```
$ cat Makefile.am
bin_PROGRAMS = hello
hello_SOURCES = hello.c
$ cat configure.in
AC_INIT(hello.c)
AC_PROG_CC
AC_PROG_INSTALL
AC_OUTPUT(Makefile)
$
$ ls -l
total 24
-rw----- 1 as  as  45 Nov 21 18:48 Makefile.am
-rw----- 1 as  as  92 Nov 21 18:49 configure.in
-rw----- 1 as  as  64 Nov 21 18:45 hello.c
$
```

Jetzt wird **autoconf** gestartet:

```
$ aclocal
$ autoconf
ls -l
$ ls -l
total 216
```

Das Shellscript **configure** ist entstanden.

```
$ ls -l
total 320
-rw----- 1 as  as      45 Nov 21 19:07 Makefile.am
-rw----- 1 as  as    28052 Nov 21 19:19 aclocal.m4
drwx----- 5 as  as     170 Nov 21 19:20 autom4te.cache
-rwx----- 1 as  as   120441 Nov 21 19:20 configure
-rw----- 1 as  as     92 Nov 21 19:19 configure.in
-rw----- 1 as  as     64 Nov 21 19:13 hello.c
$
```

Nun kann automake die erforderlichen Dateien erzeugen:

```
$ automake -a
configure.in: installing `./install-sh'
configure.in: installing `./mkinstalldirs'
configure.in: installing `./missing'
Makefile.am: installing `./INSTALL'
Makefile.am: required file `./NEWS' not found
Makefile.am: required file `./README' not found
Makefile.am: installing `./COPYING'
Makefile.am: required file `./AUTHORS' not found
Makefile.am: required file `./ChangeLog' not found
Makefile.am: installing `./depcomp'
$
```

Beim ersten Aufruf warnt automake, dass einige Dateien fehlen, wir legen sie einfach an.  
Ein erneuter Aufruf erzeugt die Datei Makefile.in.

```

$ touch NEWS README AUTHORS ChangeLog
$ automake -a
$ ls -l
total 392
-rw----- 1 as as 0 Nov 21 19:27 AUTHORS
lrwx----- 1 as as 31 Nov 21 19:23 COPYING -> /usr/share/automake-
1.6/COPYING
-rw----- 1 as as 0 Nov 21 19:27 ChangeLog
lrwx----- 1 as as 31 Nov 21 19:23 INSTALL -> /usr/share/automake-
1.6/INSTALL
-rw----- 1 as as 45 Nov 21 19:07 Makefile.am
-rw----- 1 as as 11905 Nov 21 19:27 Makefile.in
-rw----- 1 as as 0 Nov 21 19:27 NEWS
-rw----- 1 as as 0 Nov 21 19:27 README
-rw----- 1 as as 28052 Nov 21 19:19 aclocal.m4
drwx----- 5 as as 170 Nov 21 19:20 autom4te.cache
-rwx----- 1 as as 120441 Nov 21 19:20 configure
-rw----- 1 as as 92 Nov 21 19:19 configure.in
lrwx----- 1 as as 31 Nov 21 19:23 depcomp -> /usr/share/automake-
1.6/depcomp
-rw----- 1 as as 64 Nov 21 19:13 hello.c
lrwx----- 1 as as 34 Nov 21 19:23 install-sh -> /usr/share/automake-
1.6/install-sh
lrwx----- 1 as as 31 Nov 21 19:23 missing -> /usr/share/automake-
1.6/missing
lrwx----- 1 as as 37 Nov 21 19:23 mkinstalldirs -> /usr/share/automake-
1.6/mkinstalldirs

```

```
$
```

Nun ist die Installation so wie man sie dem Endbenutzer geben kann.

Das aktuelle Verzeichnis kann mittels

```
$ tar cvf hello-0.1.tar
```

erzeugt werden und per ftp (o.ä.) zur Verfügung gestellt werden.

Der Enduser kann es entpacken und mittels

```
$ ./configure  
...  
$ make  
$ make install
```

installieren.

Und mit

```
$ make uninstall
```

deinstallieren.

Eine Distribution kann man auch einfach durch

```
$ make dist
```

erstellen. Dadurch wird die Datei „hello-0.1.tar.gz“ erstellt.

Eine ausführliche Beschreibung der Beschreibungsdateien findet man z.B in:  
„<http://www.amath.washington.edu/~lf/tutorials/autoconf/toolsmanual.html>“

## **Hörsaalübung**

Erzeugen Sie eine Distribution für das Beispiel „zufall“ mit den im Skript angegebenen Dateien  
main.c, print.c, random.c, random.h

## 5. Werkzeuge zur Programmanalyse

Hier werden Werkzeuge zur

- Laufzeitanalyse und
- statischen Programmanalyse

behandelt

### 5.1. Laufzeitanalyse

Nachdem ein Programm implementiert und getestet ist, folgt meist die Phase des Tunings mit anschliessendem Endtest, bevor es zum Einsatz freigegeben wird.

Werkzeuge zur Laufzeitanalyse sind:

- **time** zur Ermittlung der **Laufzeit** von **Prozessen**
- **gprof** zeigt die **Aufteilung** der Programmlaufzeit auf die einzelnen **Funktionen**
- **gcov** zählt, **wie oft** einzelne **Programmzeilen** ausgeführt werden

## 5.1.1.time

Mittels des Kommandos `time` kann man die Ausführungszeit eines Prozesses ermitteln:

```
$ time zufall > tmp
real    4m58.811s
user    2m9.279s
sys     0m0.975s
```

Dabei sind die Werte wie folgt zu interpretieren:

`real:` tatsächlich verstrichene Zeit

`user:` CPU-Zeit, die von Benutzerfunktionen benötigt wurde

`sys:` CPU-Zeit, die von Kernelfunktionen verbraucht wurde

Diese absoluten Zahlen sind mit Vorsicht zu geniessen, denn sie sind abhängig von der Auslastung des Rechners zur Ausführungszeit und vom Zustand der Prozesse, die konkurrierend ablaufen (wegen deren Prioritäten).

Dennoch erkennt man, dass im o.a. Beispiel relativ wenig Zeit mit Kernelfunktionen gearbeitet wurde.

## 5.1.2.Gprof

Das Kommando gprof wertet Programmprofile aus, die von Programmen während der Laufzeit erstellt werden.

Vorgehen:

1. Übersetzen des zu analysierenden Programms mit der Option -pg  
Dadurch wird beim Lauf ein Profile erzeugt
2. Ausführen des Programms
3. erzeugtes Profile mit gprof auswerten

Wir nehmen unser (leicht abgewandeltes) Programm zur Erzeugung von Zufallszahlen:

```
$ cat zufall.c
#define a 100001
#define m 1717
#define c 3

void nowork() {
    long j;
    for (j=1000000; j>0; j--) ;
}

void printParameter() {
    nowork();
}
```

```

        printf("a=%ld\n", a);
        printf("c=%ld\n", c);
        printf("m=%ld\n", m);
    }

void printRandom(float i) {
    nowork();
    printf("%f\n", i);
}

float generateRandom() {
    nowork();
    static long int r=R;
    r = (a*r)%m;
    return r/(float)m;
}

main() {
    int i;
    printParameter();
    for (i=1;i<=100;i++)
        printRandom(generateRandom());
}
$

```

Die ersten 2 der o.a. Schritte:

Achtung: Compiler und Linker müssen mit `-pg` aufgerufen werden.

```
$ gcc -pg zufall.c -o zufall
$ ls -l g*
-rw-r--r--  1 as staff  534 Nov 22 15:20 gmon.out
$
```

Die Datei `gnom.out` ist das Programmprofile, das mit `gprof` ausgewertet wird (3. Schritt):

```
[as@ulab1 tmp]$ gprof zufall
Flat profile:

Each sample counts as 0.01 seconds.
 %   cumulative   self           self         total
time  seconds    seconds   calls   ms/call  ms/call  name
100.48    1.75      1.75        101    17.31    17.31   nowork
 0.00     1.75      0.00         100     0.00     0.00   generateRandom
 0.00     1.75      0.00         100     0.00    17.31   printRandom
 0.00     1.75      0.00          1     0.00    17.31   printParameter

 %
time      the percentage of the total running time of the
          program used by this function.
```

cumulative a running sum of the number of seconds accounted  
seconds for by this function and those listed above it.

self the number of seconds accounted for by this  
seconds function alone. This is the major sort for this  
listing.

calls the number of times this function was invoked, if  
this function is profiled, else blank.

self the average number of milliseconds spent in this  
ms/call function per call, if this function is profiled,  
else blank.

total the average number of milliseconds spent in this  
ms/call function and its descendents per call, if this  
function is profiled, else blank.

name the name of the function. This is the minor sort  
for this listing. The index shows the location of  
the function in the gprof listing. If the index is  
in parenthesis it shows where it would appear in  
the gprof listing if it were to be printed.

Call graph (explanation follows)

granularity: each sample hit covers 2 byte(s) for 0.57% of 1.75 seconds

| index | % time | self | children | called  | name               |
|-------|--------|------|----------|---------|--------------------|
|       |        | 0.02 | 0.00     | 1/101   | printParameter [4] |
|       |        | 1.73 | 0.00     | 100/101 | printRandom [3]    |
| [1]   | 100.0  | 1.75 | 0.00     | 101     | nowork [1]         |
| ----- |        |      |          |         |                    |
|       |        |      |          |         | <spontaneous>      |
| [2]   | 100.0  | 0.00 | 1.75     |         | main [2]           |
|       |        | 0.00 | 1.73     | 100/100 | printRandom [3]    |
|       |        | 0.00 | 0.02     | 1/1     | printParameter [4] |
|       |        | 0.00 | 0.00     | 100/100 | generateRandom [5] |
| ----- |        |      |          |         |                    |
|       |        | 0.00 | 1.73     | 100/100 | main [2]           |
| [3]   | 99.0   | 0.00 | 1.73     | 100     | printRandom [3]    |
|       |        | 1.73 | 0.00     | 100/101 | nowork [1]         |
| ----- |        |      |          |         |                    |
|       |        | 0.00 | 0.02     | 1/1     | main [2]           |
| [4]   | 1.0    | 0.00 | 0.02     | 1       | printParameter [4] |
|       |        | 0.02 | 0.00     | 1/101   | nowork [1]         |
| ----- |        |      |          |         |                    |
|       |        | 0.00 | 0.00     | 100/100 | main [2]           |
| [5]   | 0.0    | 0.00 | 0.00     | 100     | generateRandom [5] |
| ----- |        |      |          |         |                    |

Ruft printRandom auf

Wird von printRandom aufgerufen

This table describes the call tree of the program, and was sorted by

the total amount of time spent in each function and its children.

Each entry in this table consists of several lines. The line with the index number at the left hand margin lists the current function. The lines above it list the functions that called this function, and the lines below it list the functions this one called.

This line lists:

|          |   |
|----------|---|
| index    | A unique number given to each element of the table.<br>Index numbers are sorted numerically.<br>The index number is printed next to every function name so it is easier to look up where the function in the table. |
| % time   | This is the percentage of the `total' time that was spent in this function and its children. Note that due to different viewpoints, functions excluded by options, etc, these numbers will NOT add up to 100%.      |
| self     | This is the total amount of time spent in this function.  |
| children | This is the total amount of time propagated into this function by its children.   |
| called   | This is the number of times the function was called. If the function called itself recursively, the number only includes non-recursive calls, and is followed by a `+' and the number of recursive calls.           |

name           The name of the current function. The index number is printed after it. If the function is a member of a cycle, the cycle number is printed between the function's name and the index number.

For the function's parents, the fields have the following meanings:

self           This is the amount of time that was propagated directly from the function into this parent.

children       This is the amount of time that was propagated from the function's children into this parent.

called         This is the number of times this parent called the function ``/'` the total number of times the function was called. Recursive calls to the function are not included in the number after the ``/'`.

name           This is the name of the parent. The parent's index number is printed after it. If the parent is a member of a cycle, the cycle number is printed between the name and the index number.

If the parents of the function cannot be determined, the word ``<spontaneous>'` is printed in the ``name'` field, and all the other fields are blank.

For the function's children, the fields have the following meanings:

|          |   |
|----------|---|
| self     | This is the amount of time that was propagated directly from the child into the function.   |
| children | This is the amount of time that was propagated from the child's children to the function.   |
| called   | This is the number of times the function called this child `/' the total number of times the child was called. Recursive calls by the child are not listed in the number after the `/'. |
| name     | This is the name of the child. The child's index number is printed after it. If the child is a member of a cycle, the cycle number is printed between the name and the index number.    |

If there are any cycles (circles) in the call graph, there is an entry for the cycle-as-a-whole. This entry shows who called the cycle (as parents) and the members of the cycle (as children.) The `+' recursive calls entry shows the number of function calls that were internal to the cycle, and the calls entry for each member shows, for that member, how many times it was called from other members of the cycle.

```
Index by function name
```

```
[5] generateRandom
```

```
[4] printParameter
```

```
[1] nowork
```

```
[3] printRandom
```

```
$
```

### 5.1.3.gcov

Mit dem Kommando gcov (coverage) kann man eine **Abdeckungsanalyse** durchführen. Für jede Funktion wird die **Häufigkeit** von **Programmteilen** gezählt.

Das Vorgehen entspricht der Laufzeitanalyse, also:

1. Einschalten der Abdeckungsanalyse
2. Lauf des Programms mit Sammeln der Analyseergebnisse
3. Auswerten der Ergebnisse

Mittels der Optionen -fprofile-arcs und -ftest-coverage wird beim Übersetzten die Abdeckungsanalyse aktiviert.

```
$ gcc -g -fprofile-arcs -ftest-coverage -o zufall zufall.c
$ zufall > tmp
$ ls -l zu*
-rwxr-xr-x  1 as  staff 15198 Nov 22 16:28 zufall
-rw-----  1 as  staff   605 Nov 22 16:27 zufall.c
```

```
-rw-r--r--  1 as staff   348 Nov 22 16:28 zufall.gcda
-rw-r--r--  1 as staff  1444 Nov 22 16:28 zufall.gcno
$
```

Analysegraphen

Nun kann das Analyseergebniss ermittelt und angezeigt werden.

```
$ gcov zufall
File `zufall.c'
Lines executed:100.00% of 21
zufall.c:creating `zufall.c.gcov'
$
$ vi zufall.c.gcov
-:      1:
-:      2:#include <stdio.h>
-:      3:#define R 1000001
-:      4:#define a 100001
-:      5:#define m 1717
-:      6:#define c 3
-:      7:
function nowork called 101 returned 100% blocks executed 100%
    101:      8:void nowork() {
    101:      9:          long j;
101000101:     10:          for (j=1000000; j>0; )
101000000:     11:              j--;
-:      12:}
-:      13:
```

```

function printParameter called 1 returned 100% blocks executed 100%
  1:  14:void printParameter() {
  1:  15:      nowork();
  1:  16:      printf("a=%ld\n",a);
  1:  17:      printf("c=%ld\n",c);
  1:  18:      printf("m=%ld\n",m);
-:  19:}
-:  20:

function printRandom called 100 returned 100% blocks executed 100%
 100:  21:void printRandom(float i) {
 100:  22:      nowork();
 100:  23:      printf("%f\n",i);
-:  24:}
-:  25:

function generateRandom called 100 returned 100% blocks executed 100%
 100:  26:float generateRandom() {
 100:  27:      static long int r=R;
 100:  28:      r = (a*r)%m;
 100:  29:      return r/(float)m;
-:  30:}
-:  31:

function main called 1 returned 100% blocks executed 100%
  1:  32:main() {
  1:  33:      int i;
  1:  34:      printParameter();
 101:  35:      for (i=1;i<=100;i++)
 100:  36:          printRandom(generateRandom());

```

```
-: 37:}
```

Die von gcov ausgegebenen Werte bedeuten, dass 100% des Codes ausgeführt wurde.

## **5.2. Statische Programmanalyse mit splint**

## 6.rcs

rcs (Revision Controll System) ist ein System zur Versionskontrolle von Dateien.

### 6.1.Prinzip der Versionsführung

Grundidee ist es, eine **Datei in allen ihren Versionen zu verwalten**, wobei zwecks Speicherplatzeinsparung nur die Differenzen zwischen den Versionen aufbewahrt werden. Zu diesem Zweck muss irgendwo ein **Archiv** angelegt werden, entweder lokal in einem Unterverzeichnis RCS oder auf einem Server. Bei letzterem hat es sich bewährt, auf dem Server die gleiche Pfadstruktur wie auf dem lokalen Rechner zu haben. Zu diesem Zweck sollte man entsprechende kleine Skripte erstellen.

Die Grundoperationen sind nun das **Einchecken** einer Datei in das Archiv und das **Auschecken** mit und ohne **Sperre**. Eine ausgecheckte Datei darf nur dann geändert und wieder eingcheckedt werden, wenn sie mit einer Sperre (Lock) versehen wurde. Diese Sperre darf nur einmal vergeben werden, alle anderen Entwickler dürfen zwar Auschecken (z.B. um ein Projekt zu compilieren), aber nichts ändern.

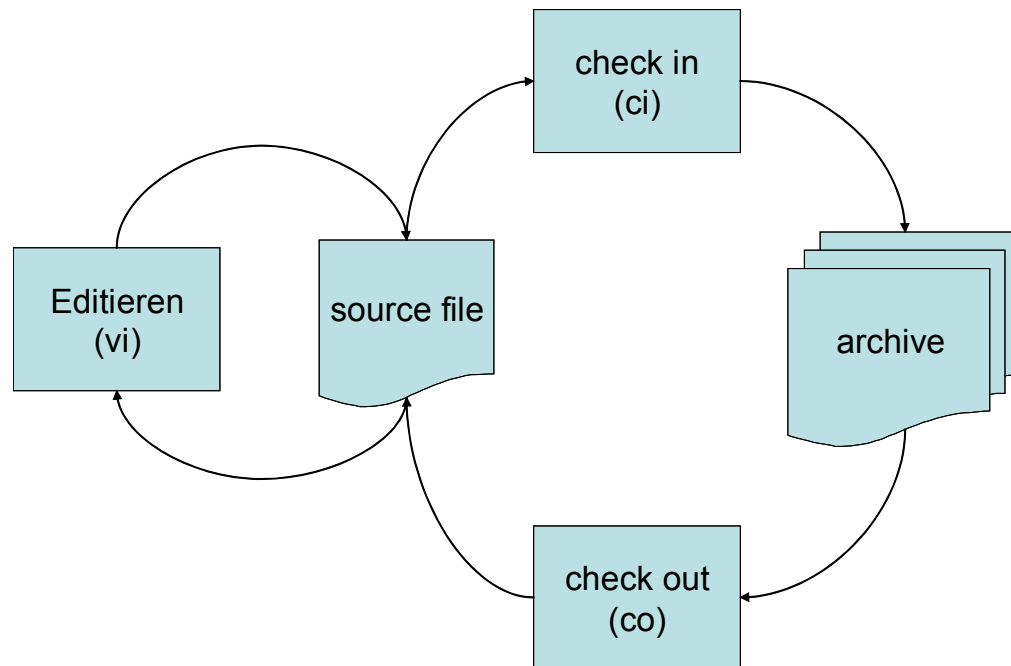
Beim Einchecken sollte man es sich zur Regel machen, nur komplette und lauffähige Versionen im Archiv abzulegen. Eine solche Version kann dann mit einem symbolischen Namen versehen werden, unter dem jederzeit wieder ein Zugriff möglich ist. Zweckmäßigerweise sollten dann natürlich alle Dateien eines Projektes diesen symbolischen Namen erhalten.

Ein weiterer wichtiger Vorteil von RCS ist der Zwang zur **Dokumentation**: beim Einchecken fragt RCS nach einem Kommentar, der die neue Version charakterisieren soll. Falls die entsprechenden RCS-Schlüsselworte im Kommentar des Dateikopfes vorhanden sind, hat man so immer einen kompletten Überblick zu allen Versionen.

Bei Problemen kann man schnell die Differenzen zwischen den Versionen anzeigen lassen, und hat somit wahrscheinliche Fehlerquellen schnell entdeckt.

Bis zur RCS Version 5.6 konnte man nur Text-Dateien verarbeiten, Binär-Dateien mussten vorher uuencode oder ähnliche Tools durchlaufen. Ab RCS 5.7 können Binär-Dateien auch direkt mit RCS verarbeitet werden.

Der Datenfluss bei der Erstellung und Wartung eines Quelltextes ist nachfolgend dargestellt.



Das Modul wird vom Programmierer erstellt. Dies kann in mehreren Editor-Sitzungen geschehen (vi).

Das Modul wird vom Programmierer bezüglich seiner Schnittstellenspezifikation getestet und verändert, bis es die gewünschten Eigenschaften besitzt.

Das "fehlerfreie" Modul wird vom Programmierer in der Bibliothek des Versionskontrollsystems gesichert (ci). Das Modul ist eingefroren. Erst jetzt steht das Modul anderen Programmierern des Teams bzw. zum Integrationstest zur Verfügung.

Irgendwann wird das Modul in einem Integrationstest innerhalb des Projektes getestet. Sollten dabei Fehler in diesem Modul gefunden werden, müssen diese behoben werden. Dazu wird der betroffene Programmierer das Modul aus der Bibliotheksdatei entnehmen (co) und die notwendigen Änderungen vornehmen. Nach Test wird es erneut eingefroren (ci).

Wenn das Programm bereits einige Zeit in Gebrauch ist, kann es nötig werden, an einigen Modulen Erweiterungen vorzunehmen, um das Produkt weiterzuentwickeln. Zu diesem Zweck muss das Modul ebenfalls der Versionsbibliothek entnommen werden (co).

Zusammenfassend ist festzuhalten, dass Quelltexte beliebig oft editiert werden können, bis sie mit einem speziellen Kommando im Versionskontrollsystem eingefroren werden. Bei der Änderung eines eingefrorenen Moduls wird automatisch eine neue Version erzeugt, die wiederum beliebig oft verändert werden kann, bis auch sie eingefroren wird. Man wird stets nur solche Versionen einfrieren, die bereits einen Spezifikationstest bestanden haben. Weiterhin müssen man ein Modul einfrieren, bevor andere Mitglieder des Projektteams es verwenden können und dürfen! Sicherlich gibt es außerhalb des Versionskontrollsystems weitere Möglichkeiten, Quelltexte und Module auszutauschen. Dennoch sollten Sie ausschließlich Module anderer Programmierer Ihres Teams benutzen, die (automatisch) dem Versionskontrollsystem entnommen wurden. Nur so können Sie bei einem Integrationstest die Übersicht behalten und sicher sein, immer die rich-

tigen Versionen zu verwenden. Das Versionskontrollsystem ist also gleichzeitig Archiv und Schnittstelle zwischen den einzelnen Programmierern.

## 6.2.Beispielsitzung

Gehen wir von folgendem Programm aus:

```
$ cat hello.c
#include <stdio.h>
main()
{
    printf("Hello, rcs.\n");
}
$
```

Das Einfrieren wird durch folgendes Kommando durchgeführt, dabei wird nach einer Beschreibung der Version gefragt:

```
$ ci hello.c
hello.c,v <-- hello.c
initial revision: 1.1
enter description, terminated with ^D or '.':
NOTE: This is NOT the log message!
>> Erste Version
>>
done
```

```
$
```

Jetzt ist eine rcs Datei erzeugt worden, die „hello.c,v“ heist; die Datei „hello.c“ wurde gelöscht. In der rcs Datei ist die Datei hello.c in der Version 1.1 eingefroren.

Das Wiederherstellen der letzten Version geschieht durch:

```
$ co hello.c
hello.c,v --> hello.c
revision 1.1
done
$
$ ls -l hello.c
-r--r--r--  1 swta10      a1          57 Nov 20 15:53 hello.c
$
```

Die Datei darf nicht verändert werden.

Soll eine Änderung an der Datei hello.c vorgenommen werden, muss man eine Arbeitsversion erzeugen:

```
$ co -l hello.c
hello.c,v --> hello.c
revision 1.1 (locked)
done
$ ls -l
```

```
total 46
-rwxrwxrwx   1 swta10      a1          20533 Nov 20 15:39 hello
-rw-r--r--   1 swta10      a1             57 Nov 20 15:59 hello.c
-r--r--r--   1 swta10      a1          285 Nov 20 15:59 hello.c,v
$
```

Durch das Locking der Datei ist sicher gestellt, dass niemand anderes ein check in vornehmen kann.

Nun kann die Änderung vorgenommen werden:

```
$ cat hello.c
#include <stdio.h>
main()
{
    printf("Hello, rcs.\nErste Aenderung.\n");
}
$
```

Nach dem Test des Programms wird die zweite Version eingefroren.

```
$ ci hello.c
hello.c,v <-- hello.c
new revision: 1.2; previous revision: 1.1
```

```
enter log message:
(terminate with ^D or single '.')
>> Zweite Version
>>
done
$
```

Der **Unterschied** zwischen zwei Versionen ist durch folgendes Kommando ersichtlich:

```
$ rcsdiff -r1.1 -r1.2 hello.c
RCS file: hello.c,v
retrieving revision 1.1
retrieving revision 1.2
rdiff -r1.1 -r1.2
4c4
<     printf("Hello, rcs.\n");
---
>     printf("Hello, rcs.\nErste Aenderung.\n");
$
```

In Quelldateien kann man besondere Strings einbetten. Dadurch ist durch das Kommando ident ein einfaches „**Dokumentationssystem**“ verfügbar. Dazu muss lediglich die Zeile

```
char rcsid[]="$Header: $" ;
```

in die Quelldatei eingebettet sein. Die Informationen werden dann automatisch durch ci in der Quelldatei ersetzt. Zudem ist dann auch die Dokumentation in der ausführbaren Datei enthalten.

```
$ cat hello.c
#include <stdio.h>
char rcsid[]="$Header: hello.c,v 1.3 01/11/20 16:15:32 swta10 Exp $" ;
main()
{
    printf("Hello, rcs.\nErste Aenderung.\n");
}
$
```

```
$ ci -l hello.c
hello.c,v <-- hello.c
new revision: 1.3; previous revision: 1.2
enter log message:
(terminate with ^D or single '.')
>> Letzte Version
>>
done
$ make hello
```

```
    cc -O hello.c -o hello
$ ident hello
hello:
    $Revision: 92453-07 linker linker crt0.o B.10.32 990409 $
    $Header: hello.c,v 1.3 01/11/20 16:15:32 swta10 Exp $
$
```

Durch das Kommando „rcs“ kann man die Dateiattribute eines rcs Files ändern.