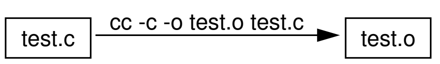
D 2. Übung

D.1 Überblick

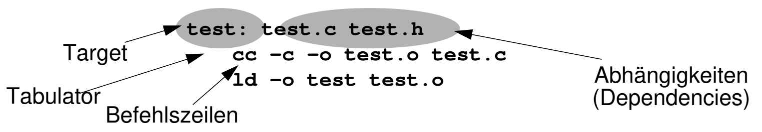
- Übersetzen von Projekten mit "make"
- Versionverwaltung mit RCS
- Aufgabe 2: qsort

D.2 Make

- Problem: Es gibt Dateien, die aus anderen Dateien generiert werden.
  - ◆ Zum Beispiel kann eine test.o Datei aus einer test.c Datei unter Verwendung des C-Compilers generiert werden.



- Ausführung von *Update*-Operationen
- **`Makefile`**: enthält Abhängigkeiten und Update-Regeln (Befehlszeilen)



1 Beispiel

```
test: test.o func.o
    ld -o test test.o func.o

test.o: test.c test.h func.h
    cc -c test.c

func.o: func.c func.h test.h
    cc -c func.c
```


2 Allgemeines

- Kommentare beginnen mit **`#`** (bis Zeilenende)
- Befehlszeilen müssen mit TAB beginnen
- das zu erstellende Target kann beim **`make`**-Aufruf angegeben werden (z.B. **`make test`**)
  - ◆ wenn kein Target angegeben wird, bearbeitet make das erste Target im Makefile
- beginnt eine Befehlszeile mit **`@`** wird sie nicht ausgegeben
- jede Zeile wird mit einer neuen Shell ausgeführt (d.h. z.B. **`cd`** in einer Zeile hat keine Auswirkung auf die nächste Zeile)

3 Makros

- in einem Makefile können Makros definiert werden

```
SOURCE = test.c func.c
```

- Verwendung der Makros mit `$(NAME)` oder `${NAME}`

```
test: $(SOURCE)
    cc -o test $(SOURCE)
```


4 Dynamische Makros

- `$@` Name des Targets

```
test: $(SOURCE)
    cc -o $@ $(SOURCE)
```

- `$*` Basisname des Targets

```
test.o: test.c test.h
    cc -c $*.c
```

- `$?` Abhängigkeiten, die jünger als das Target sind
- `$<` Name einer Abhängigkeit (in impliziten Regeln)

5 ... Makros

- Erzeugung neuer Makros durch Konkatenation

```
OBJS += hallo.o
```

oder

```
OBJS = $(OBJS)  hallo.o
```

- Erzeugen neuer Makros durch Ersetzung in existierenden Makros

```
OBJS_SOLARIS = $(OBJS:test.o=test_solaris.o)
```

- Ersetzen mit Pattern-Matching

```
SOURCE = test.c func.c
OBJS = $(SOURCE:%.c=%.o)
```

- Benutzen von Befehlsausgaben

```
WORKDIR = $(shell pwd)
```


6 Eingebaute Regeln und Makros

- make enthält eingebaute Regeln und Makros (`make -p` zeigt diese an)
- Wichtige Makros:
  - ◆ `CC` C-Compiler Befehl
  - ◆ `CFLAGS` Optionen für den C-Compiler
  - ◆ `LD` Linker Befehl
  - ◆ `LDFLAGS` Optionen für den Linker
- Wichtige Regeln:
  - ◆ `.c.o` C-Datei in Objektdatei übersetzen
  - ◆ `.c` C-Datei übersetzen und linken

7 Suffix Regeln

- Eine Suffix Regel kann verwendet werden, wenn **make** eine Datei mit einer bestimmten Endung (z.B. **test.o**) benötigt und eine andere Datei gleichen Namens mit einer anderen Endung (z.B. **test.c**) vorhanden ist.

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

- Suffixe müssen deklariert werden

```
.SUFFIXES: .c .o $(SUFFIXES)
```

- Explizite Regeln überschreiben die Suffix-Regeln

```
test.o: test.c
    $(CC) $(CFLAGS) -DXYZ -c $<
```


8 Beispiel verbessert

```
SOURCE = test.c func.c
OBJS = $(SOURCE:%.c=%.o)
HEADER = test.h func.h

test: $(OBJS)
    @echo Folgende Dateien erzwingen neu-linken von $@: $?
    $(LD) $(LDFLAGS) -o $@ $(OBJS)

.c.o:
    @echo Folgende C-Datei wird neu uebersetzt: $<
    $(CC) $(CFLAGS) -c $<


test.o: test.c $(HEADER)

func.o: func.c $(HEADER)
```


9 Nützliche Konvention

- Aufräumen mit **make clean**

```
clean:
    rm -f $(OBJS)
```

- Projekt bauen mit **make all**

```
all: test
```

- Installieren mit **make install**

```
install: all
    cp test /usr/local/bin
```


D.3 Revision Control System – RCS

1 Einführung

- RCS ist ein Versionskontrollsystem, das
  - ◆ Änderungen an Dateien mit dem Namen des Ändernden, dem Zeitpunkt und einem Kommentar speichert
  - ◆ Zugriffe auf Versionen kontrolliert und koordiniert
  - ◆ eindeutige Identifizierung verwendeter Versionen erlaubt
  - ◆ redundante Speicherung von Versionen vermeidet
    - ➥ es wird jeweils die letzte Version einer Datei gespeichert
    - ➥ zusätzlich werden sog. *reverse deltas* (Beschreibungen, wie aus Version n Version n-1 erzeugt wird) abgelegt

2 Einführung (2)

- RCS besteht aus einer Reihe von Kommandos, die es dem Benutzer erlauben
  - ◆ Dateien unter RCS-Kontrolle zu stellen und Kopien aller Versionen zu bekommen, die danach erstellt wurden
  - ◆ eine Version zum Editieren zu entnehmen und diese gegen gleichzeitige Änderungen zu sperren
  - ◆ Neue Versionen (mit Kommentar) zu erzeugen
  - ◆ Unbrauchbare Änderungen rückgängig zu machen
  - ◆ Zustandsinformation von Dateien abzufragen
    - ➤ Zeilenweise Unterschiede zwischen verschiedenen Versionen auszugeben
    - ➤ Log-Informationen über Versionen: Urheber, Datum, usw.

3 Terminologie

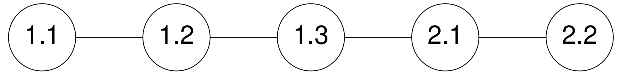
- Delta
  - ◆ Menge von zeilenweisen Änderungen an der Version einer Datei unter der Kontrolle von RCS
    (die Begriffe “Version" und “Delta” werden oft synonym gebraucht)
- Revision-Id
  - ◆ Jede Version erhält zur Identifikation eine Identifikation zugewiesen:

    `Release-Nummer.Level-Nummer`
- RCS-Datei
  - ◆ enthält die neueste Version und alle vorhergehenden Versionen in Form von Deltas zusammen mit Verwaltungsinformationen
  - ◆ der Dateiname endet auf **`,v`**, die RCS-Datei ist entweder im Unterdirectory **`RCS`**, oder im gleichen Directory wie die Arbeitsdatei abgelegt
- Arbeitsdatei
  - ◆ Kopie einer Version aus der RCS-Datei

4 Nummerierung von Versionen

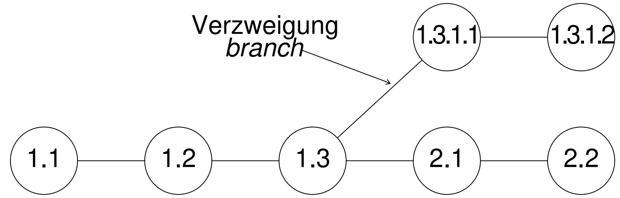
- Versionen werden ausgehend von der Ur-Version nummeriert:

  `release.level`



- Versionen in einer Verzweigung erhalten

  `release.level.branch.branchlevel`



5 Kommandos — Überblick

ci(1)	***check in*** speichert die Arbeitsdatei als neue Version in der RCS-Datei ab falls noch nicht vorhanden, wird ein neue RCS-Datei erzeugt
co(1)	***check out*** extrahiert eine existierende Version aus der RCS-Datei (nur zum Lesen oder exklusiv zum Schreiben)
rcs(1)	Modifikation von RCS-Datei-Attributen
rlog(1)	Ausgabe von *log*-Information und RCS-Datei-Attributen
ident(1)	extrahiert RCS-Identifikatoren aus einer Datei
rcsclean(1)	nicht-modifizierte Arbeitsdateien löschen
rcsdiff(1)	*diff* zwischen Versionen einer RCS-Datei
rcsmerge(1)	erzeugt aus zwei Versionen (insbes. bei Verzweigungen) eine neue Version

- bei allen Kommandos kann als ***filename*** immer sowohl der Arbeitsdateiname oder der RCS-Dateiname angegeben werden

5 Kommandos — *ci(1)*

- *check in RCS-Revisions* — Erzeugen neuer Versionen
  - ◆ ***ci(1)*** übernimmt neue Versionen in RCS-Dateien
  - ◆ die neue Version wird aus der jeweiligen Arbeitsdatei entnommen, die Arbeitsdatei wird anschließend gelöscht
  - ◆ existierte zu der Arbeitsdatei noch keine RCS-Datei, wird eine neue RCS-Datei erzeugt
- Aufrufsyntax (nur die wichtigsten Optionen angegeben!):

  `ci [-rrev] [-lrev] [-urev] filename ...`

  `-r`*rev* die neue Version erhält Version ***rev***
  – ***rev*** muß größer als die letzte existierende Version sein
  – soll eine neue *Release* erzeugt werden, genügt die Angabe der *Release*-Nummer (z. B. **`-r5`**)

  `-l`*rev* wie **`ci -r`**, anschließend wird automatisch ein **`co -l`** durchgeführt

  `-u`*rev* wie **`ci -r`**, anschließend erfolgt ein **`co`**

5 Kommandos — *ci(1)*

- Beispiel ***ci, rlog***

```
% ci prog.c
RCS/prog.c,v  <--  prog.c
initial revision: 1.1
enter description, terminated with single '.' or end of file:
NOTE: This is NOT the log message!
>> Program to demonstrate RCS
>> .
done
% rlog prog.c

RCS file: RCS/prog.c,v
Working file: prog.c
head: 1.1
branch:
locks: strict
access list:
symbolic names:
comment leader: " * "
keyword substitution: kv
total revisions: 1;     selected revisions: 1
description:
Program to demonstrate RCS
----------------------------
revision 1.1
date: 1992/07/20 11:56:43;  author: jklein;  state: Exp;
Initial revision
=============================================================
%
```


5 Kommandos — *co(1)*

- ◆ *check out RCS Revisions* — Versionen entnehmen
- ***co(1)*** entnimmt eine Version aus allen angegebenen RCS-Dateien
- die entnommene Version wird als Arbeitsdatei abgespeichert
- der Name der Arbeitsdatei ergibt sich aus dem Namen der RCS-Datei, wobei die Endung **`,v`** und ggf. der Pfad-Prefix **`RCS/`** weggelassen werden
  - ◆ Aufrufsyntax (nur die wichtigsten Optionen angegeben!):

  `co [-rrev] [-lrev] [-urev] filename ...`

  `-r`*rev* extrahiert die neueste Version, der Versionsnummer kleiner oder gleich ***rev*** ist

  `-l`*rev* wie **`co -r`**, die extrahiert Version wird durch den Aufrufer gesperrt

  `-u`*rev* wie **`co -r`**, falls eine Sperre der Version durch den Aufrufer existiert, wird diese aufgehoben

5 Kommandos — *co(1)*

- Beispiel ***co, rlog***

```
% co -l prog.c
RCS/prog.c,v  -->  prog.c
revision 1.1 (locked)
done
% rlog prog.c

RCS file: RCS/prog.c,v
Working file: prog.c
head: 1.1
branch:
locks: strict
        jklein: 1.1
access list:
symbolic names:
comment leader: " * "
keyword substitution: kv
total revisions: 1;     selected revisions: 1
description:
Program to demonstrate RCS
----------------------------
revision 1.1    locked by: jklein;
date: 1992/07/20 11:56:43;  author: jklein;  state: Exp;
Initial revision
=============================================================
%
```

6 Identifikation von RCS-Versionen

- RCS ersetzt bei einem ***check out*** im Text alle Vorkommen der Zeichenkette
  `$Id$`
  durch
  `$Id:` *filename revisionnumber date time author state locker*

- ***co(1)*** sorgt dafür, daß diese Zeichenkette automatisch auf aktuellem Stand gehalten wird

- um diese Zeichenkette in Objekt-Code zu implantieren, reicht es, sie in als *String* im Programm anzugeben — in C z. B.
  `static char rcsid[] = "$Id$";`

- mit dem Kommando ***ident(1)*** können solche RCS-Identifikatoren aus beliebigen Dateien extrahiert werden
  - ➥ damit ist z. B. feststellbar, aus welchen Versionen der Quelldateien ein ausführbares Programm entstanden ist

D.4 Aufgabe 2: Sortieren mittels qsort

1 Funktion *qsort(3)*

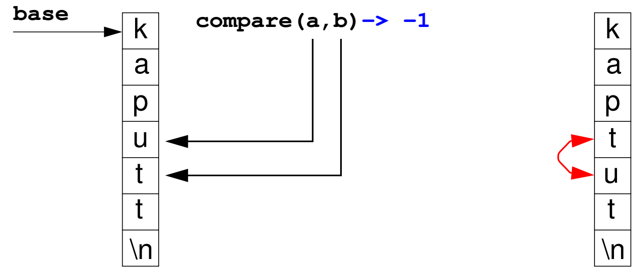
- Prototyp aus **`stdlib.h`**:

```
void qsort(void *base,
       size_t nel,
       size_t width,
       int (*compare) (const void *, const void *));
```

- Bedeutung der Parameter:
  - ◆ **base** : Zeiger auf das erste Element des Feldes, dessen Elemente sortiert werden sollen
  - ◆ **nel** : Anzahl der Elemente im zu sortierenden Feld
  - ◆ **width**: Größe eines Elements
  - ◆ **compare**: Vergleichsfunktion

2 Arbeitsweise von *qsort(3)*

- ◆ **qsort** vergleicht je zwei Elemente mit Hilfe der Vergleichsfunktion compare
- ◆ sind die Elemente zu vertauschen, dann werden die entsprechenden Felder komplett ausgetauscht, z.B.:



3 Vergleichsfunktion

- Die Vergleichsfunktion erhält Zeiger auf Feldelemente, d.h. die übergebenen Zeiger haben denselben Typ wie das Feld

- Die Funktion vergleicht die beiden Elemente und liefert:
  - <0, falls Element 1 kleiner bewertet wird als Element 2
  - 0, falls Element 1 und Element 2 gleich gewertet werden
  - >0, falls Element 1 größer bewertet wird als Element 2

- Beispiel:
  - ◆ 'z', 'a' -> 1
  - ◆ 1, 5 -> -1
  - ◆ 5,5 -> 0