Einführung zur Aufgabengruppe 2

- **Aufbau eines C-Programms nach ANSI-Standard**
- **Erzeugen eines ausführbaren Programms**
- **Programmerstellung unter UNIX**
- **Das Konfigurationswerkzeug make, Konzeption und seine Funktionsweise**
- **Aufbau eines Makefiles**

Aufbau eines C-Programms nach ANSI-Standard, erläutert am Vorlesungsbeispiel

Allgemeine Hinweise finden Sie im Vorlesungskapitel 8 zur Einführung in die Programmiersprache C im Rahmen des Propädeutikums für die Studienanfänger im WS (http://www.informatik.uni-leipzig.de/~meiler/).

Ein C-Programm liegt in der Regel auf mehreren Quelldateien vor. Das Programm zum Vorlesungsbeispiel ist in die folgenden Module unterteilt.

Mehrere Module

Modul **scalar.c**:

Direktiven für den Präprozessor
globale Deklarationen

main(. . .)
{ *lokale Deklarationen*
Anweisungsfolge
}

Modul **io.c**:

Direktiven für den Präprozessor
globale Deklarationen

f1(. . .)
{ *lokale Deklarationen*
Anweisungsfolge
}

io.h:

Direktiven für den Präprozessor
globale Deklarationen

Modul **vec.c**:

Direktiven für den Präprozessor
globale Deklarationen

f2(. . .)
{ *lokale Deklarationen*
Anweisungsfolge
}

. . .

fn(. . .)
{ *Anweisungsfolge*
lokale Deklarationen
}

vec.h:

Direktiven für den Präprozessor
globale Deklarationen

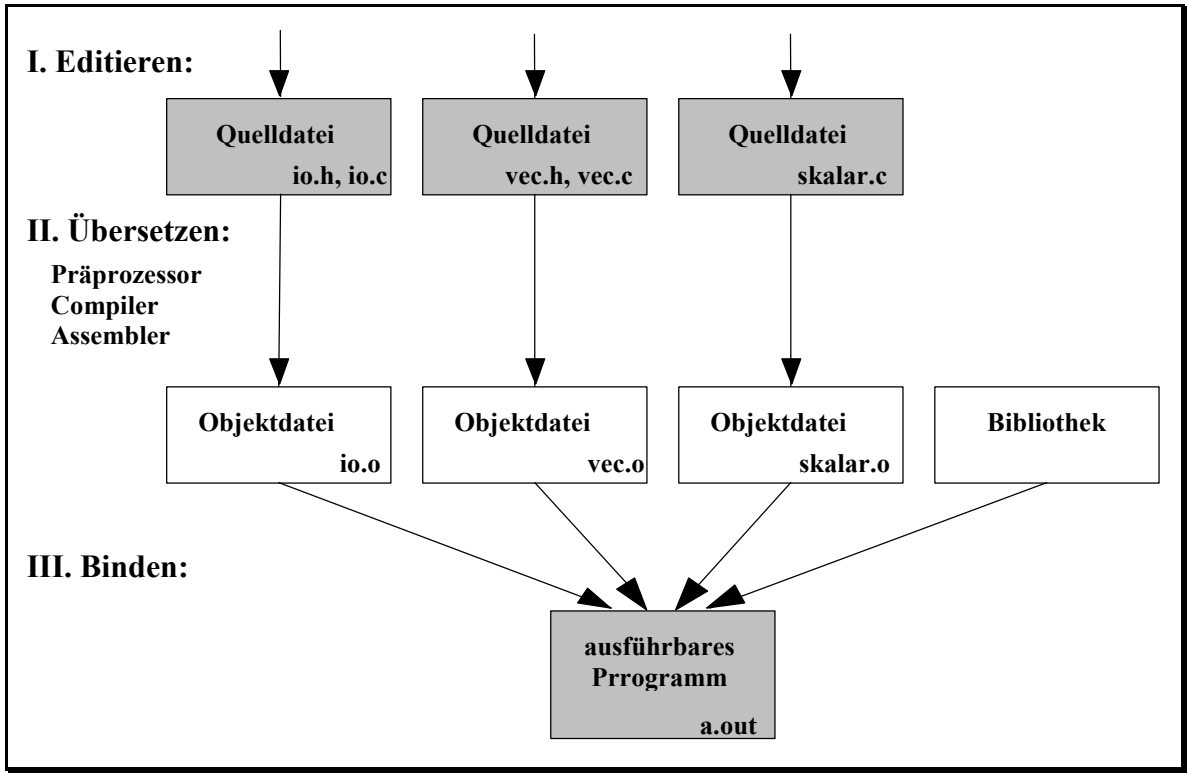
Erzeugen eines ausführbaren Programms

Bei der Erstellung eines ausführbaren Programms sind **drei Schritte** notwendig.

I. Editieren: Schreiben des Quelltextes mit Hilfe eines beliebigen vorhandenen Editors.

II. Übersetzen: Jede zu einem C-Programm gehörige Quelldatei wird in drei Phasen zu einer Objektdatei übersetzt (Präprozessor, Compiler, Assembler).

III. Binden: Die Objektdateien werden zusammen mit den benötigten Bibliotheksdateien zu einem ausführbaren Programm gebunden.



Für das Vorlesungsbeispiel sieht das folgendermaßen aus:

zu I:
```
$ nedit scalar.c &
$ nedit io.h &
$ nedit io.c &
$ nedit vec.h &
$ nedit vec.c &
```

zu II und III:
```
$ gcc scalar.c vec.c io.c
```

Ausführen:
```
$ a.out
```

Programmerstellung unter UNIX

Zu I. **Editoren:**

vi	(UNIX-Standardeditor)
nedit	(UNIX-Editor mit grafischer Oberfläche)
emacs	(GNU-Editor)
xwpe	(Programmieroberfläche)

Zu II./III. **Übersetzen/Binden:**

$ cc [*optionen* ***] main.c modul11.c modul22.c . . . [*** *optionen* ***]*** (UNIX-Standard)

$ gcc [*optionen* ***] main.c modul11.c modul22.c . . . [*** *optionen* ***]*** (GNU – C)

optionen

Compiler:	**-P (**gcc:-**E)**	Nur Präprozessorlauf: **prog.i**.
	-S	Präprozessor- und Compilerlauf: **prog.s**.
	-c	Übersetzen ohne Binden: **prog.o**.
	-O	Übersetzen mit Optimierung.
	-ansi	Ansi-Standard
	-Wall	Alle Warnungen werden angezeigt.
	-g	Debuggerinformationen (Generierung der Symboltabellen)
	-I*include.dir*	Suchpfad für Include-Dateien (Standard: /**usr/include**).
Binder:	**-l***libery*	Bibliothek *libery* ist aus dem Standard-Bibliotheks-Verzeichnis dazuzubinden. In **/lib** und **/usr/lib** wird die Bibliothek **lib***library***.a** gesucht. Standard-C-Funktionen **libc.a** sind stets geladen.
	-L*library.dir*	Suchpfad für Bibliotheken (Standard: **/lib** und **/usr/lib**).
	-o *name*	Das ausführbare Programm heißt *name*, sonst **a.out**.

Vorlesungsbeispiel:

$ gcc scalar.c vec.c io.c ⇒ ***a.out***

besser:

$ gcc -Wall -O -ansi scalar.c vec.c io.c -o scalar ⇒ ***scalar***

noch besser:

$ gcc -Wall -O -ansi -c io.c ⇒ ***io.o***

$ gcc -Wall -O -ansi -c vec.c ⇒ ***vec.o***

$ gcc -Wall -O -ansi -c scalar.c ⇒ ***scalar.o***

$ gcc io.o vec.o scalar.o -o scalar ⇒ ***scalar***

Anderes Beispiele:

$ gcc -Wall -O -ansi kurve.c -lm -o kurve ⇒ ***kurve***

(Bibliothek **/usr/lib/libm.a** für mathematische Funktionen aus **/usr/include/math.h**)

$ gcc -Wall -O -ansi -I/dargb/pub/graf/vogle/vogle-1.3.0/local/include kurven.c k.c
-L/dargb/pub/graf/vogle/vogle-1.3.0/local/lib -lvogle -lX11 -lm -o kurven

⇒ ***kurven***

(Bibliothek **/dargb/pub/graf/vogle/vogle-1.3.0/local/lib/libvogle.a** für Grafik-Funktionen aus **/dargb/pub/graf/vogle/vogle-1.3.0/local/include /vogle.h**)

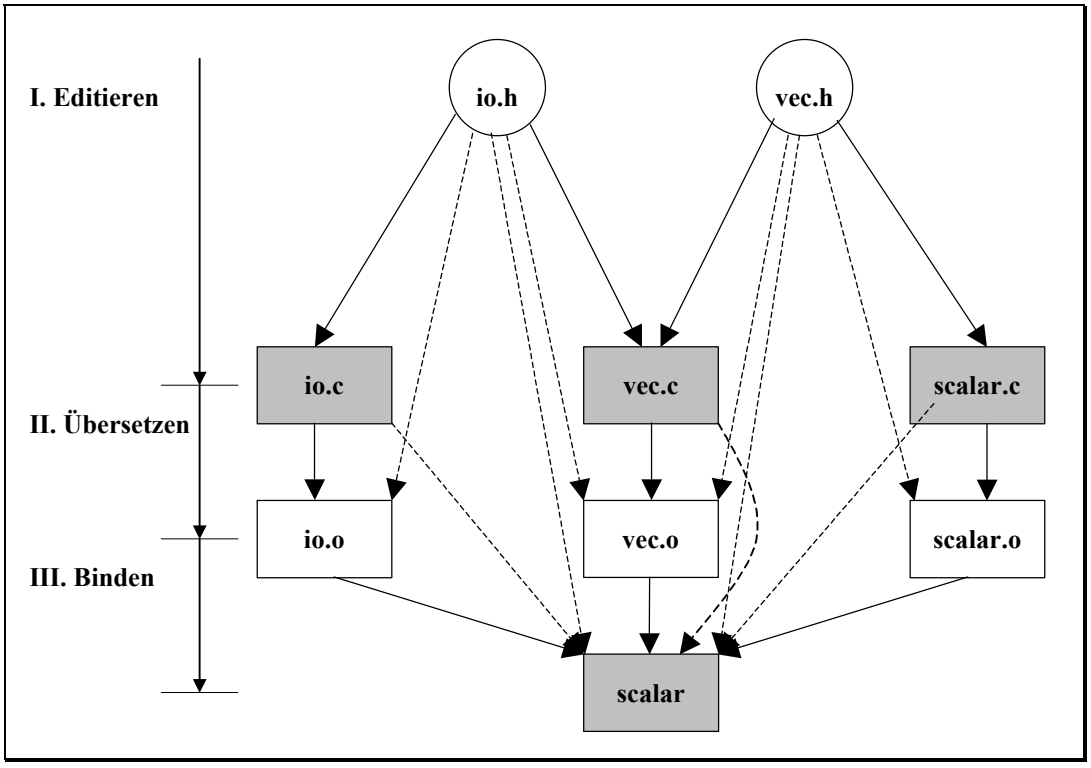
Das Konfigurationswerkzeug make, Konzeption und seine Funktionsweise

make ist ein Unixwerkzeug zum Ausführen von Aktionen (wie Übersetzen, Binden, Drucken) anhand von **Zeitstempeln** und **Abhängigkeiten** zwischen verschiedenen Modulen und beteiligten Dateien. Zur Steuerung der Erzeugung eines ausführbaren Programms dient eine spezielle Datei mit dem Namen **makefile** und einer standardmäßig gegebene Konfigurationsdatei **make.cfg**.

make kennt den **Zeitstempel** (Zeitpunkt der letzten Änderung) ⇒ Betriebssytem
make kennt nicht die **Abhängigkeiten** zwischen den Objektdateien ⇒ **makefile**
make kennt nicht die **Kommandos** zur Generierung eines Moduls ⇒ **makefile**

Abhängigkeiten und Kommandos

Beispiel: Ableitungsbaum des Vorlesungsbeispiels.



1. Version: Vollständige Darstellung in einer Datei **makefile**.

Kommentar

```
#################################################################
# makefile f"ur das Programm scalar (einfach)              MM #
#################################################################

scalar : scalar.c vec.c vec.h io.c io.h                 Abhängigkeiten
TAB gcc –Wall –O -ansi scalar.c vec.c io.c -o scalar    Kommando

testein: testein.c io.c io.h
```

```
TAB gcc –Wall –O -ansi testein.c io.c -o testein

testvec: testvec.c vec.c vec.h io.c io.h
TAB gcc –Wall –O -ansi testvec.c vec.c io.c -o testvec

testscal: testscal.c vec.c vec.h io.c io.h
TAB gcc –Wall –O -ansi testscal.c vec.c io.c -o testscal
```

$ make ⇒ **gcc -Wall -O -ansi scalar.c vec.c io.c -o scalar** *Protokoll*
$ make (ohne erneute Änderungen) ⇒ **‘ scalar ‘ is up to date.**
$ make testein ⇒ **gcc -Wall -O -ansi testein.c io.c -o testein**

Makrodefinitionen

Syntax: *Makroname = Zeichenfolge*
Aufruf: **$ (** *Makroname* **)**

2. Version: Datei **makefile** mit Makros.

```
#############################################################
# makefile f"ur das Programm scalar (einfach)            MM #
#############################################################

CC = gcc
CFLAGS = -Wall -O -ansi
LDFLAGS = -o
OUT = scalar testein testvec testscal

scalar : scalar.c vec.c vec.h io.c io.h
TAB $(CC) $(CFLAGS) scalar.c vec.c io.c $(LDFLAGS) scalar

testein: testein.c io.c io.h
TAB $(CC) $(CFLAGS) testein.c io.c $(LDFLAGS) testein

testvec: testvec.c vec.c vec.h io.c io.h
TAB $(CC) $(CFLAGS) testvec.c vec.c io.c $(LDFLAGS) testvec

testscal: testscal.c vec.c vec.h io.c io.h
TAB $(CC) $(CFLAGS) testscal.c vec.c io.c $(LDFLAGS) testscal

clean:
TAB rm -f $(OUT)
```

$ make ⇒ Warten des Programms **scalar**
$ make testscal ⇒ Warten des Programms **testscal**
$ make clean ⇒ Löschen von **scalar, testein, testvec, testscal** nach Rückfrage

3. Version: Datei **makefile** mit Makros und Erzeugung von Objektdateien.

```
#################################################################
# makefile f"ur das Programm scalar (besser)              MM #
#################################################################

CC = gcc
CFLAGS = -Wall –O –ansi -c
LDFLAGS = -o
OUT = scalar testein testvec testscal

# Binden von Objektcode

scalar: scalar.o vec.o io.o
TAB $(CC) scalar.o vec.o io.o $(LDFLAGS) scalar

testscal: testscal.o vec.o io.o
TAB $(CC) testscal.o vec.o io.o $(LDFLAGS) testscal

testvec: testvec.o vec.o io.o
TAB $(CC) testvec.o vec.o io.o $(LDFLAGS) testvec

testein: testein.o io.o
TAB $(CC) testein.o io.o $(LDFLAGS) testein

# Erzeugen des Objektcode

scalar.o: scalar.c vec.h
TAB $(CC) $(CFLAGS) scalar.c

vec.o: vec.c vec.h io.h
TAB $(CC) $(CFLAGS) vec.c

io.o: io.c io.h
TAB $(CC) $(CFLAGS) io.c

testscal.o: testscal.c vec.h
TAB $(CC) $(CFLAGS) testscal.c

testvec.o: testvec.c vec.h
TAB $(CC) $(CFLAGS) testvec.c

testein.o: testein.c io.h
TAB $(CC) $(CFLAGS) testein.c

clean:
TAB rm -f $(OUT)
```

$ make ⇒ **gcc -Wall -O -c -ansi io.c** *Protokoll*
⇒ **gcc -Wall -O -c -ansi vec.c**
⇒ **gcc -Wall -O -c -ansi scalar.c**
⇒ **gcc scalar.o vec.o io.o -o scalar**

$ make (nur **io.c** wurde geändert, **io.c** in Abhängigkeit von **io.o**)

⇒ **gcc -Wall -O -c -ansi io.c**

⇒ **gcc scalar.o vec.o io.o -o scalar**

4. Version: Datei **makefile** mit Makros, Erzeugung von Objektdateien und Druckservice.

```
#############################################################
# Dieses makefile wartet das Programm scalar              MM #
# und druckt geaenderte Dateien aus.                        #
#############################################################

CC = gcc                                          Makrodefinitionen
CFLAGS = -Wall –O –ansi –c
LDFLAGS = -o
SRC = scalar.c vec.c io.c
OBJ = scalar.o vec.o io.o
OUT = scalar testein testvec testscal
FILES=$(SRC) io.h vec.h makefile                      geschachtelt

all: scalar print                                 Abhängigkeiten

# Binden von Objektcode

scalar: $(OBJ)
TAB $(CC) $(OBJ) $(LDFLAGS) scalar

testscal: testscal.o vec.o io.o
TAB $(CC) testscal.o vec.o io.o $(LDFLAGS) testscal

testvec: testvec.o vec.o io.o
TAB $(CC) testvec.o vec.o io.o $(LDFLAGS) testvec

testein: testein.o io.o
TAB $(CC) testein.o io.o $(LDFLAGS) testein

# Erzeugen des Objektcode

scalar.o: scalar.c vec.h
TAB $(CC) $(CFLAGS) scalar.c

vec.o: vec.c vec.h io.h
TAB $(CC) $(CFLAGS) vec.c

io.o: io.c io.h
TAB $(CC) $(CFLAGS) io.c

testscal.o: testscal.c vec.h
TAB $(CC) $(CFLAGS) testscal.c

testvec.o: testvec.c vec.h
TAB $(CC) $(CFLAGS) testvec.c

testein.o: testein.c io.h
```

```
TAB $(CC) $(CFLAGS) testein.c

clean:
TAB rm -f $(OUT) $(OBJ)

# Druckkommando, $? geänderte Dateien,
# -l72 Seitenlaenge, -n mit Zeilennummerierung,
# touch Zeitstempel der Datei print der Laenge 0

print: $(FILES)
TAB print -l72 -n $? | lpr
TAB touch print
```

$ make scalar	⇒	Warten des Programms **scalar**
$ make print	⇒	Drucken aller nach dem letzten Druck veränderten Dateien
$ make	⇒	beides

Interne Makros (dynamisch evaluiert)

$@	Vollständiger Name des Zielobjekts.
$*	Name des Zielobjekts ohne Suffix.
$?	Namen der Dateien, die neuer sind, als die Zieldatei.
$<	Name der Datei, die aktuell verarbeitet wird.

$(*D), $(@D), $(<D)	s.o. einschließlich Pfad, evtl. „.“
$(*F), $(@F), $(<F)	s.o. ohne Pfad

Produktionsregeln

Um Abhängigkeiten bei der Anwendung von Produktionsregeln zu beschreiben, ist nur der Suffix von Bedeutung.

Beispiel: ***$ make scalar.o***

scalar.c → **scalar.o** ⇒ Produktionsregel: **.c.o:**

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

Produktionsregeln ermöglichen allgemeingültige Kommandodefinitionen. Eine Auswahl ist in der Datei **make.cfg** (/usr/ccs/lib/make.cfg) voreingestellt. Wir gehen hier auf diese nicht ein.

Ein weiteres Beispiel

Einbinden der Vogle-Bibliothek (s.o.):

```
#################################################################
# Dieses makefile wartet das Programm kurven.            MM #
#################################################################

CC = gcc
CFLAGS = -Wall –O –ansi –c
LDFLAGS = -lm –lvogle –lX11 -o
OBJ = kurven.o k.o
LIBS = -L/dargb/pub/graf/vogle/vogle-1.3.0/local/lib
INC = -I/dargb/pub/graf/vogle/vogle-1.3.0/local/include

kurven: $(OBJ)
TAB $(CC) $(OBJ) $(LIBS) $(LDFLAGS) kurven

kurven.o: kurven.c k.h
TAB $(CC) $(INC) $(CFLAGS) kurven.c

k.o: k.c k.h
TAB $(CC) $(INC) $(CFLAGS) k.c

clean:
TAB rm -f $(OBJ)
```

$ make ⇒ **kurven**