10 Präprozessor und eigene Bibliotheken

10.1 Zum Präprozessor

Alle Zeilen, deren erstes sichtbares Zeichen `#` ist, sind Anweisungen für den Präprozessor. Die Zeilen können mit einem \ fortgesetzt werden. Die Hauptanwendungen des Präprozessors sind das Einfügen von Dateien, die Definition von Makros und die bedingte Übersetzung.

Einfügen von Dateien:

#include <*datei*> diese Zeile wird durch den Inhalt von *datei*, die in dem Verzeichnis `/usr/include` gesucht wird, ersetzt. Präprozessor in *datei* werden befolgt. Vorsicht beim mehrfachen Einbinden von Dateien.

Einfügen von Dateien:

#include "*datei*" diese Zeile wird durch den Inhalt von *datei*, die in dem aktuellen Verzeichnis, dann in `/usr/include` gesucht wird, ersetzt. Präprozessoranweisungen in *datei* werden befolgt.

Einfache Makros:

#define *makro ersatz* im Programm wird *makro* durch *ersatz* ersetzt, wenn *makro* als eigenständiges Wort auftritt. Bei Teilworten und innerhalb von Zeichenketten erfolgt kein Ersatz. Präprozessoranweisungen selbst bleiben unverändert.

Beispiel:

```
#define INT long int
#define DBLE long double
```

Es wird *INT* durch `long int` ersetzt, sowie *DBLE* durch `long double`.

Makros mit Argumenten:

#define *makro*(*parameter*) *ersatz* im Programm wird *makro* durch *ersatz* ersetzt, wenn *makro* als eigenständiges Wort auftritt. Hierbei werden die Parameter berücksichtigt. Wenn nötig können Definitionen mit `#undef` gelöscht werden.

```
#include <stdio.h>

#define summe(X,Y) X+Y
#define rsumme(X,Y) ((X)+(Y))

int main(void){
int a, b;
a = summe(4,3)*summe(4,3);
b = rsumme(4,3)*rsumme(4,3);
printf("%d %d \n", a, b);
return 0;
}
```

Vorsicht bei den Parametern: `summe(4,3)*summe(4,3)` wird mit `4+3*4+3` ersetzt. Durch das Klammern kann man diese Effekte in den Griff bekommen.
Steht ein `#` vor einem Parameter *par* im Ersetzungstext, so wird *par* durch "*par*" ersetzt. Wenn in *par* Anführungszeichen oder Backslashes vorkommen, werden sie mit vorangestelltem Metazeichen Backslash versehen.
Steht ein `##` vor oder hinter einem Parameter *par* im Ersetzungstext, so wird werden die `##` inklusive ihrer benachbarten blanks und Tabulatoren fortgelassen. So kann man z.B. zwei Variablen hintereinanderhängen.
Beispiele:

```
#define QUADR(A) (A)*(A)
#define CAT(A,B) A ##  B
#define MAXWORT 1949
```

Im Programm wird aus *cat(MAX, WORT)* zunächst *MAXWORT*, das wird durch `1949` ersetzt.

```
#define MAX(A,B) ( (A) > (B) ) ? (A) : (B)
```

Aufheben von Makros:

#undef *makro* ab jetzt wird *makro* nicht mehr ersetzt.

If-Abfragen:

#ifdef *var*
#statement1
#else
#statement2
#endif Wenn *var* definiert ist, wird *statement1* ausgeführt, sonst *statement2*.

#ifndef *var*
#statement1
#else
#statement2
#endif Wenn *var* nicht definiert ist, wird *statement1* ausgeführt, sonst *statement2*.

#if *condition1*
#statement1
#elif *condition2*
#statement2
#else #statement3 #endif Wenn *condition1* erfüllt ist, wird *statement1* ausgeführt, wenn *condition2* erfüllt ist, *statement2*, sonst *statement3*.

Fehlermeldungen:

#error "*Meldung*" Der Präprozessor gibt *Meldung* aus.

Beispiele:

```
#define OS linux
#ifdef OS
#include "linux.h"
#else
#include "sun.h"
#endif

#ifndef MACRO
#error "MACRO nicht definiert"
#endif
```

```
#include <stdio.h>

int main(void){
int i=1;
#if i == 1
printf("i hat den Wert 1\n");
#else
printf("i ist ungleich 1\n");
#endif
return 0;
}
```

10.2 Optionen für den Compiler

gcc *-options progs.c progs.o* Es können mehrere `.c`- bzw `.o`-Dateien verwendet werden.

wichtige Optionen		
`-c`	kein linken	erzeugt *prog*`.o`
`-E`	nur Präprozessor	Info auf console
`-S`	nur Assembler code	erzeugt *prog*`.s`
`-o name`	Name der ausführbaren Datei	erzeugt *name*
`-g`	code zum debuggen mit `gdb` oder `xxgdb`	
`-pg`	code zum profiling mit `gprof`	
`-ansi -pedantic`	erlaubt nur ANSI-Stil	
`-l`*bib*	bindet auch `libbib` ein	
`-L`*path*	sucht unter *path* nach Bibliotheken	
`-Wall`	viele Meldungen beim Übersetzen	
`-O`*zahl*	Optimierungsstufe	

10.3 Grundlegendes zu make

Um ein größers Programm, dessen Funktionen in eigenen Datein stehen, besser warten und erweitern zu können, verwendet man `make`. Hiermit kann man erreichen, daß nur die Programmteile neu übersetzt werden müssen, die seit dem letzten `make` verändert wurden.

make [**-f** ***file***] führt die Anweisungen in der Datei `Makefile` aus, wahlweise kann mit der Option `-f` ein anderer Dateiname gewählt werden.

Die Anweisungen in der Datei `Makefile` bestehen aus der Angabe von Abhängigkeiten und Compileranweisungen. Mit Hilfe von Variablen kann man die Schreibarbeit systematisieren. Prinzipiell haben die Anweisungen folgende Form:

Zieldatei:	Dateien die hierzu vorliegen müssen
(target):	(dependencies)
TAB	Anweisungen

Die folgenden Beispiele sollen das einfache Arbeiten demonstrieren. Wir gehen von den Programmen

```
komplex.c
cadd.c ccabs.c cconj.c cdiv.c cmult.c csub.c
complex.h
```

aus, mit denen wir komplexe Zahlen eingeführt haben. `main` ist in `komplex.c` enthalten, die restlichen .c Dateien enthalten die Funktionen, die für komplexe Zahlen geschrieben wurden, `complex.h`, schließlich, enthält eine eigene header-Datei, die von allen Quellprogrammen beötigt wird.

```
# erster Versuch in der Datei Makefile
# Aufruf: make clean
#bzw.
#   make

komplex: komplex.o cadd.o ccabs.o cconj.o cdiv.o cmult.o csub.o
        gcc -o komplex cadd.o ccabs.o cconj.o cdiv.o cmult.o csub.o komplex.o
clean:
        rm -f *.o

cadd.o: cadd.c complex.h Makefile
        gcc -c cadd.c
ccabs: ccabs.c complex.h Makefile
        gcc -c ccabs.c
cconj: cconj.c complex.h Makefile
        gcc -c cconj.c
cdiv.o: cdiv.c complex.h Makefile
        gcc -c cdiv.c
cmult.o: cmult.c complex.h Makefile
        gcc -c cmult.c
csub.o: csub.c complex.h Makefile
        gcc -c csub.c
komplex.o: komplex.c complex.h Makefile
        gcc -c komplex.c
```

```
# zweiter Versuch in der Datei Makefile1
# Aufruf: make -f Makefile1 clean
#bzw.
#   make -f Makefile1
#
CC = gcc
OBJ = komplex.o cadd.o ccabs.o cconj.o cdiv.o cmult.o csub.o
CFLAGS = -Wall -ansi -pedantic

komplex: $(OBJ)
        $(CC) $(CFLAGS) -o komplex $(OBJ)
clean:
        rm -f *.o

cadd.o: cadd.c complex.h Makefile1
        $(CC) -c $(CFLAGS) cadd.c
ccabs: ccabs.c complex.h Makefile1
        $(CC) -c $(CFLAGS) ccabs.c
cconj: cconj.c complex.h Makefile1
        $(CC) -c $(CFLAGS) cconj.c
cdiv.o: cdiv.c complex.h Makefile1
        $(CC) -c $(CFLAGS) cdiv.c
cmult.o: cmult.c complex.h Makefile1
        $(CC) -c $(CFLAGS) cmult.c
csub.o: csub.c complex.h Makefile1
        $(CC) -c $(CFLAGS) csub.c
komplex.o: komplex.c complex.h Makefile1
        $(CC) -c $(CFLAGS) komplex.c
```

```
# dritter Versuch in der Datei Makefile2
# Aufruf: make -f Makefile2 clean
#bzw.
#   make -f Makefile2
#
CC = gcc
OBJ = cadd.o ccabs.o cconj.o cdiv.o cmult.o csub.o komplex.o
CFLAGS = -Wall -ansi -pedantic

komplex: $(OBJ)
        $(CC) $(CFLAGS) -o komplex $(OBJ)
clean:
        rm -f *.o komplex

cadd.o: complex.h Makefile2
ccabs: complex.h Makefile2
cconj: complex.h Makefile2
cdiv.o: complex.h Makefile2
cmult.o: complex.h Makefile2
csub.o: complex.h Makefile2
komplex.o: complex.h Makefile2
```

10.4 Eigene Bibliotheken

Wir führen an einem Beispiel das Arbeiten mit Bibliotheken vor.
Datei: dvector.c

```
double *dvector(int low,int high) {


double *v;
v = (double *) calloc((size_t) (high -low+1),sizeof(double));
if (!v)
        printf("allocation failure in dvector()\n");
v = v - low;
return v;
}
```

Datei: printdvector.c

```
void printdvector(int n, double *v, char c) {
int i;
printf("%c = (");
for (i = 1; i <= n; i++)
        printf("%lf, ",v[i]);
printf("\b\b)\n");
}
```

Datei: vectors.c

```
#include<stdio.h>
int main() {
double *v;
int i, n;
printf("Eingabe n: ");
scanf("%d", &n);
v = dvector(1,n);
for (i = 1; i <= n; i++) {
v[i] = 1.0;
}
printdvector(n,v,'v');
return 0;
}
```

Wir führen folgende Schritte durch:

- `gcc -c` *dvector.c* – erzeugt `.o`-Datei,
- `gcc -c` *printdvector.c* – erzeugt `.o`-Datei,
- `ar r` *mylib printdvector.o* – ersetzt in *mylib* die Datei *printdvector.o*.
- `ar d` *mylib printdvector.o* – löscht in *mylib* die Datei *printdvector.o*.

Damit haben wir eine eigene Bibliothek *mylib* angelegt. Analog speichern wir u.U. die Quellen in *mylib.src*.

- `ar q mylib` legt die Bibliothek *mylib* an.
- `ar r` *mylib.src dvector.c*
- `ar r` *mylib.src printdvector.c*

In einem *readme* könnte man sich noch die Funktionsheader abspeichern, also:

```
double *dvector(int low,int high)
dynamisches Anlegen eines double Vektors der Dimension high-low+1
void printdvector(int n, double *v, char c)
Drucken eines double Vektors mit Namen c
```

und diese Datei ebenfalls in *mylib.src* ablegen.
Folgende Befehle sind nützlich.

- `ar t` *mylib* – Inhaltsverzeichnis,
- `ar p` *mylib.src printdvector.c* – Kopie von *printvector.c* auf den Bildschirm.
- `ar x` *mylib printdvector.o* – Kopie von *printvector.o* in das aktuelle Verzeichnis.

Das Hauptprogramm kann man nun mit
`gcc` *vectors.c -ovectors mylib*
übersetzen.

```
#  make -f Makefile3
CC = gcc
OBJ = cadd.o ccabs.o cconj.o cdiv.o cmult.o csub.o
CFLAGS = -Wall -ansi -pedantic
komplex: libmy.a $(OBJ)
        $(CC) $(CFLAGS) komplex.c libmy.a -o komplex
#       $(CC) $(CFLAGS) komplex.c -L$(PWD) -lmy -o komplex
clean:
        rm -f *.o
cadd.o: complex.h Makefile3 cadd.c
        $(CC) -c cadd.c
        ar rs libmy.src cadd.c
        ar rs libmy.a cadd.o
ccabs.o: complex.h Makefile3 ccabs.c
        $(CC) -c ccabs.c
        ar rs libmy.src ccabs.c
        ar rs libmy.a ccabs.o
cconj.o: complex.h Makefile3 cconj.c
        $(CC) -c cconj.c
        ar rs libmy.src cconj.c
        ar rs libmy.a cconj.o
cdiv.o: complex.h Makefile3 cdiv.c
        $(CC) -c cdiv.c
        ar rs libmy.src cdiv.c
        ar rs libmy.a cdiv.o
cmult.o: complex.h Makefile3 cmult.c
        $(CC) -c cmult.c
        ar rs libmy.src cmult.c
        ar rs libmy.a cmult.o
csub.o: complex.h Makefile3 csub.c
        $(CC) -c csub.c
        ar rs libmy.src csub.c
        ar rs libmy.a csub.o
```