Inhaltsverzeichnis

C-Programme mit dem gcc bearbeiten

1. Der gcc

Das Programm `gcc` ist eine Art (nicht-graphische) *Benutzeroberfläche* oder *Facade* (engl. frontend) für zahlreiche Programme, mit denen man aus *Quelldateien* eine *ausführbare Datei* erzeugen kann. Z.B. kann man mit dem `gcc` aus einer C-Quelldatei `Hallo.c` eine ausführbare Datei namens `Hallo.exe` erzeugen. Zu den Programmen, die man mit Hilfe des `gcc` aufrufen kann, gehören *Compiler* für verschiedene Sprachen (Fortran, C, C++, ObjectiveC, Pascal, Ada, Java, ...), ein sogenanntes *Binderprogramm* (engl. *binder* oder *linkage editor*) und weitere Werkzeuge.

Solange alles glatt läuft braucht ein Benutzer meistens nicht genau zu wissen, welche Werkzeuge (Compiler, Binder etc.) vom `gcc` aufgerufen werden. Aber spätestens wenn die Werkzeuge *Fehlermeldungen* ausgeben ist es günstig zu wissen, von wem die Meldungen stammen und in welchem Zusammenhang die gemeldeten Fehler aufgetreten sind. Diese Einführung soll den Zusammenhang beschreiben, den der `gcc` schafft.

Zum Namen `gcc`: Unter Unix hieß (und heißt) der C-Compiler üblicherweise `cc` (der noch kürzere Name `c` wurde verworfen, weil ein versehentlicher Druck auf die Taste C keine Compilation starten soll). Der vom Gnu-Projekt entwickelte C-Compiler wurde deshalb `gcc` genannt (Gnu C-Compiler). Als der `gcc` immer mehr zu einer Benutzeroberfläche für mehrere Compiler und andere Werkzeuge wurde, führte man eine neue Deutung der Abkürzung `gcc` ein: *gnu compiler collection*. Das Programm `gcc` ruft heute unter anderem den Gnu-C-Compiler `cc` und den Binder `ld` auf.

Ob es im Deutschen *der* `gcc` (wie „der Compiler“, *die* `gcc` (wie „die Kollektion“ oder „die Benutzeroberfläche“) oder *das* `gcc` (wie „das Werkzeug“) heissen sollte, ist nicht endgültig geklärt. In diesem Papier wird *der* `gcc` verwendet.

2. Woraus besteht ein C-Programm?

Auf diese Frage gibt es mehrere gute Antworten. Eine davon lautet etwa so:

Ein C-Programm besteht aus *Quelldateien* (engl. source files). Man unterscheidet zwei Arten von Quelldateien: *Implementierungsdateien* (z.B. `Hallo.c`) und *Kopfdateien* (engl. header files, z.B. `stdlib.h`). Im einfachsten Fall besteht ein C-Programm aus einer einzigen Implementierungsdatei.

Genau eine der Implementierungsdateien eines C-Programms muss eine `main`-Funktion enthalten. Wir bezeichnen diese Datei hier auch als die *Hauptdatei* des Programms.

Anmerkung: Eine andere gute Antwort auf die Frage nach den Bestandteilen eines C-Programms beginnt etwa so: Ein C-Programm besteht aus Vereinbarungen. Man unterscheidet zwei Arten von Vereinbarungen: Deklarationen und Definitionen. ... etc. Eine dritte Antwort beginnt so: Ein C-Programm besteht aus Lexemen, die von einem Lexer in Token umgewandelt werden ... etc. Im Folgenden werden wir aber nur von der *ersten* Antwort ausgehen.

3. Von den Quelldateien zu einer ausführbaren Datei

Quelldateien kann man (unabhängig von der verwendeten Programmiersprache) auf ganz verschiedene Weise ausführbar machen bzw. ausführen. Eine besonders einfache Weise besteht darin, die Quelldateien direkt (ohne Übersetzungen oder andere Zwischenschritte) von einem (Quellcode-) *Interpreter* ausführen zu lassen (siehe z.B. `www.softintegration.com`). Mit der Sprache Java hat sich dagegen ein ziemlich komplexes Übersetzungs-und-Ausführungsmodell verbreitet, bei dem 2 verschiedene Compiler und 2 verschiedene Interpreter an der Übersetzung und Ausführung eines Programms beteiligt sind.

C-Programme werden in aller Regel nach einem relativ alten und bewährten ("klassischen") Übersetzungs-und-Ausführungsmodell übersetzt und ausgeführt. Auch der `gcc` arbeitet nach diesem klassischen Modell, welches im Folgenden anhand von Beispielen erläutert wird.

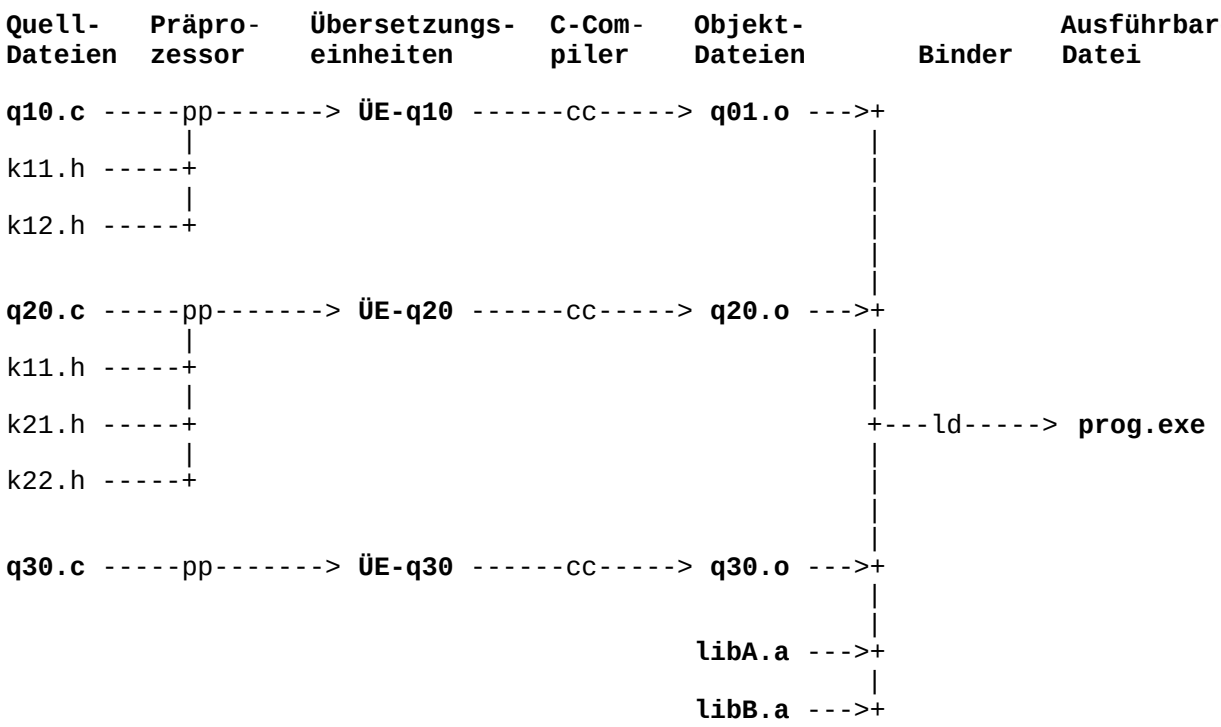
Beispiel-01: Erzeugung einer ausführbaren Datei aus sieben Quelldateien
Angenommen, ein C-Programm besteht aus den folgenden Quelldateien:

4 Kopfdateien:	`k11.h`, `k12.h`, `k21.h`, `k22.h`
3 Implementierungsdateien:	`q10.c` (inkludiert `k11.h` und `k12.h`) `q20.c` (inkludiert `k11.h`, `k21.h` und `k22.h`) `q30.c` (inkludiert keine Kopfdateien)

Man beachte, dass in diesem Beispiel die Kopfdatei `k11.h` von `q10.c` *und* von `q20.c` inkludiert wird. Im allgemeinen kann eine Kopfdatei von beliebig vielen Quelldateien (außer von sich selbst) inkludiert werden. Die Implementierungsdatei `q30.c` inkludiert in diesem Beispiel keine Kopfdatei. Das kommt in der Praxis eher selten vor, ist aber durchaus erlaubt.

Das klassische Kompilations-und-Ausführungsmodell sieht *drei* Werkzeuge vor: Einen *Präprozessor*, einen *Compiler* und einen *Binder*. Das folgende Diagramm soll die Arbeitsschritte dieser Werkzeuge illustrieren:

Diagramm-01: Erzeugung einer ausführbaren Datei `prog.exe` aus sieben Quelldateien (`q10.c`, `q20.c`, `q30.c`, `k11.h`, `k12.h`, `k21.h`, `k22.h`):



Die einzelnen Teile dieses Diagramms, insbesondere die *Übersetzungseinheiten* ÜE-q10, ÜE-q20, ÜE-q30 und die *Bibliotheken* libA.a, libB.a werden in den folgenden Abschnitten genauer erläutert.

4. Der Präprozessor übersetzt Implementierungsdateien in Übersetzungseinheiten

Jede einzelne *Implementierungsdatei* (.c-Datei) wird vom Präprozessor in eine *Übersetzungseinheit* übersetzt. Dabei werden alle Präprozessor-Befehle ausgeführt. Diese Befehle erkennt man daran, dass sie mit einem Nummernzeichen # beginnen (z.B. #ifdef oder #define oder #include etc.).

Unter den Präprozessor-Befehlen sind die #include-Befehle besonders wichtig. Jeder include-Befehl muss den Namen einer Kopfdatei (.h-Datei) enthalten und wird vom Präprozessor durch *den Inhalt dieser Kopfdatei* ersetzt.

Im Beispiel soll die Datei q10.c die folgenden beiden include-Befehle enthalten:

```
#include <k11.h>
#include <k12.h>
```

Etwas vereinfacht gesagt ist die Übersetzungseinheit ÜE-q10 eine Kopie der Datei q10.c, aber anstelle der beiden include-Befehle enthält sie den Inhalt der beiden Kopfdateien k11.h und k12.h.

Da der Präprozessor nicht nur die include-Befehle, sondern auch alle anderen Präprozessorbefehle ausführt, kann die Übersetzungseinheit ÜE-q10 sich noch an anderen Stellen von der Datei q10.c unterscheiden.

Übersetzungseinheiten haben keine offiziellen Namen (die Namen ÜE-q10, ÜE-q20 etc. wurden nur für diese Darstellung "frei erfunden") und ihren Inhalt zeigt einem der gcc nur auf ausdrücklichen Wunsch, wie etwa im folgenden Beispiel.

Beispiel-01: Übersetzungseinheiten ausgeben lassen. Die gcc-Optionen -E und -C

```
 1 > gcc -E q10.c
 2 > gcc -E -C q20.c > KarlHeinz
```

Erläuterung: Die Zeilenanfänge 1>, 2> etc. sollen eine Zeilen-Nummer und ein Promptzeichen > darstellen und müssen (und dürfen) nicht eingegeben werden. Die eigentlichen Kommandos beginnen hier erst nach dem Promptzeichen mit gcc -E ... .

Der Befehl in Zeile 1 gibt die Übersetzungseinheit ÜE-q10 zur Standardausgabe (zum Bildschirm) aus. Der Befehl in Zeile 2 schreibt die Übersetzungseinheit ÜE-q20 in eine Datei namens Karl-Heinz. Dort kann man sie dann mit einem Editor ansehen. Die Option -C bewirkt, dass auch Commentare aus der Quelldatei in die Übersetzungseinheit kopiert werden (sonst werden sie nicht kopiert).

Merkregel: Eine *Implementierungsdatei* (z.B. q10.c) kann Präprozessorbefehle enthalten. Eine *Übersetzungseinheit* (z.B. ÜE-q10) enthält *keine* Präprozessorbefehle mehr (möglicherweise aber ihre "Wirkungen und Ergebnisse").

Aufgabe-01: Schreiben Sie ein möglichst einfaches Hallo-Programm in eine Quelldatei namens Hallo.c und lassen Sie sich die Übersetzungseinheit ÜE-Hallo ausgeben (wie im Beispiel-01). Aus wieviel Zeilen besteht die Übersetzungseinheit ÜE-Hallo? Erstaunlich, oder?

Manchmal ist es wichtig zu wissen, *woher* Kopfdateien (engl. header files) stammen. Der Präprozessor hat eine Liste von Verzeichnissen, in denen er nach Kopfdateien (die er inkludieren soll) sucht. Normalerweise enthält diese Kopfdatei-Verzeichnisliste mindestens all die Verzeichnisse, in denen die Standardkopfdateien (stdlib.h, stdio.h etc.) stehen.

Beispiel-02: Die Kopfdatei-Verzeichnisliste ausgeben lassen. Die Option -v

```
 3 > gcc -v -c q10.c
```

Die Option -v (wie "verbose", geschwätzig) veranlaßt den gcc, viele informative Meldungen auszugeben, darunter auch die Liste der Verzeichnisse, in denen der Präprozessor nach Kopfdateien sucht.

Eine Kopfdatei `k0.h` kann andere Kopfdateien `k1.h`, `k2.h`, ... etc. inkludieren. Wenn eine Implementierungsdatei eine solche Kopfdatei `k0.h` *direkt* inkludiert (d.h. mit einem `#include`-Befehl), inkludiert sie damit *indirekt* auch die Kopfdateien `k1.h`, `k2.h`, ... etc.

Die `gcc`-Optionen `-M` und `-MM` sind eigentlich für die automatische Erzeugung von `make`-Dateien gedacht. Man kann mit ihnen aber auch feststellen, welche Kopfdateien von einer Implementierungsdatei direkt oder indirekt inkludiert werden, etwa so:

Beispiel-03: Welche Kopfdateien werden von der Implementierungsdatei `q10.c` (direkt oder indirekt) inkludiert? Die `gcc`-Optionen `-M` und `-MM`

```
 4 > gcc -M q10.c
 5 > gcc -MM q10.c
```

Das Kommando in Zeile 4 gibt (u.a.) die Pfadnamen *aller* von `q10.c` direkt oder indirekt inkludierten Kopfdateien aus, auch die der Standard-Kopfdateien (wie `stdlib.h` und `stdio.h` etc., siehe Anhang B). Häufig interessiert man sich aber nur für die "selbsgeschriebenen Kopfdateien". Das Kommando in Zeile 5 gibt nur die Pfadnamen von *nicht-Standard-Kopfdateien* aus.

In einem Aufruf des `gcc` kann man mit der Option `-I` (wie "include") weitere Verzeichnisse angeben, in denen der Präprozessor nach Kopfdateien suchen soll.

Beispiel-04: Weitere Kopfdatei-Verzeichnisse angeben. Die `gcc`-Option `-I`

Angenommen, die Implementierungsdatei `q20.c` beginnt wie folgt:

```
 6 // Datei q20.c
 7 #include <stdlib.c>  // Spitze Klammern             <...>
 8 #include <k21.h>     // Spitze Klammern             <...>
 9 #include "k22.h"     // Doppelte Anfuehrungszeichen "..."
10 ...
```

Sei weiter angenommen, dass wir die Datei `q20.c` mit folgendem Kommando compilieren lassen:

```
11 > gcc -c -I/projekt/kv1 -I/immer/kv q20.c
```

Dann gilt: Der Präprozessor trägt die Verzeichnisse `/projekt/kv1` und `/immer/kv` in seine Kopfdatei-Verzeichnisliste ein und schreibt sie dort *vor* all die Verzeichnisse, in denen die Standardkopfdateien stehen.

Weil die Kopfdateien `stdlib.h` und `k21.h` (siehe Zeile 7 und 8) in spitzen Klammern `<...>` angegeben wurden, werden sie *in allen Kopfdatei-Verzeichnissen* gesucht.

Weil die Kopfdatei `k22.h` (siehe Zeile 9) in doppelten Anführungszeichen `"..."` angegeben wurde, sucht der Präprozessor sie zuerst im *aktuellen Arbeitsverzeichnis.* Falls er sie dort nicht findet, sucht er sie auch in allen *Kopfdatei-Verzeichnissen.*

Um bestimmte Fehlermeldungen zu verstehen, sollte man wissen: Wenn der Präprozessor einen `include`-Befehl durch den Inhalt einer Kopfdatei ersetzt, achtet er *nicht* auf irgendwelche syntaktischen oder anderen Regeln, er kopiert einfach. Man kann sich das auch so merken: Der Präprozessor hat keine Ahnung von den Regeln, die man beim Schreiben eines C-Programms beachten muss und warnt einen nicht vor Fehlern in Kopfdateien. Besonders unangenehm ist das z.B. dann, wenn eine Kopfdatei mit einem *unvollständigen Befehl* endet, etwa so:

Beispiel-05: Fehler am Ende einer Kopfdatei (Teil A)

```
12 // Datei k11.h
13 ...
14 ...
15 int summe(int n1, int n2)
```

Am Ende der Funktionsdeklaration in Zeile 15 fehlt ein Semikolon. Das merkt der Präprozessor aber nicht. Wenn man die Kopfdatei `k11.h` in einer Implementierungsdatei `q10.c` inkludiert, etwa so:

Beispiel-06: Fehler am Ende einer Kopfdatei (Teil B)

```
16 // Datei q10.c
17 #include <k11.h>
18
19 int main() {
20    ...
21 }
```

meldet der Compiler einen Fehler, aber nicht in Zeile 17 (wo die fehlerhafte Kopfdatei inkludiert wird), sondern erst in Zeile 19. Denn man hätte das fehlende Semikolon ja nicht unbedingt in Zeile 15 der Kopfdatei `k11.h` notieren müssen, sondern hätte es auch in der Implementierungsdatei `q10.c` in Zeile 18 oder am Anfang von Zeile 19 (vor `int main()` ...) "nachliefern" dürfen. Weil das Semikolon fehlt, hält der Compiler die `main`-Funktion für einen (falschen) Teil der Funktionsdeklaration in Zeile 15 und gibt eine nicht ganz einfach zu verstehende Fehlermeldung aus.

Die Standard-Kopfdateien wie `stdlib.h` und `stdio.h` etc. (siehe Anhang B) sind ziemlich gut ausgetestet und enthalten mit hoher Wahrscheinlichkeit keine solchen Fehler mehr. Aber wenn man selbstgeschriebene Kopfdateien verwendet und unverständliche Fehlermeldungen des Compilers bekommt, sollte man die Kopfdateien noch einmal prüfen, auch wenn sie in der Fehlermeldung nicht erwähnt werden.

5. Der C-Compiler übersetzt Übersetzungseinheiten in Objektdateien

Jede einzelne Übersetzungseinheit (z.B. `ÜE-q10`) muss mit dem C-Compiler in eine *Objektdatei* (`q10.o`) übersetzt werden. Das gilt zumindest konzeptuell (d.h. "so sollte man es sich vorstellen").

Praktisch kann man dem `gcc` (und nicht direkt dem C-Compiler) befehlen, eine oder mehrere Implementierungsdatei(en) zu compilieren, etwa so:

Beispiel-01: Eine Implementierungsdatei in eine Objektdatei übersetzen. Die `gcc`-Option `-c`

```
1 > gcc -c q10.c
```

Dieses Kommando bewirkt, dass der `gcc` den C-Compiler aufruft. Der C-Compiler ruft dann automatisch den Präprozessor auf. Der Präprozessor erzeugt aus der Implementierungsdatei `q10.c` die Übersetzungseinheit `ÜE-q10` und übergibt sie dem Compiler. Der Compiler prüft die Übersetzungseinheit und meldet entweder Fehler oder übersetzt die Einheit in eine Objektdatei namens `q10.o`.

Merke: Die Bezeichnung *Objektdatei* ist deutlich älter als die Methode der *objektorientierten Programmierung* und hat nichts mit ihr zu tun.

Merke: Übersetzungseinheitein haben keine offiziellen Namen. Aber eine Implementierungsdatei und ihre Objektdatei haben normalerweise *gleiche Namen* und nur *unterschiedliche Erweiterungen* (`.c` bzw. `.o`). Wenn der Programmierer ihm nicht ausdrücklich etwas anderes befiehlt, erzeugt der `gcc` aus einer Implementierungsdatei namens `KarlHeinz.c` eine Objektdatei namens `KarlHeinz.o`.

Beispiel-02: Einer Objektdatei ausdrücklich einen bestimmten Namen geben

```
2 > gcc -c KarlHeinz.c -o KlausDieter.abc
```

Dieses Kommando befiehlt dem `gcc`, aus der Quelldatei `KarlHeinz.c` eine Objektdatei namens `KlausDieter.abc` zu erzeugen. Solche nicht-Standard Namen für Objektdateien sind aber nur selten nützlich.

Die folgende Regel ist besonders wichtig, um das klassische C-Übersetzungs-und-Ausführungsmodell genauer zu verstehen:

Die Kein-Gedächtnis-Regel: Während der C-Compiler eine Übersetzungseinheit (z.B. `ÜE-q10`) bearbeitet, betrachtet er nur *diese eine Einheit.* Er holt sich keine weiteren Informationen aus anderen Dateien oder sonstigen Quellen, um z.B. zu prüfen, ob die aktuelle Einheit `ÜE-q10` mit anderen Einheiten "zusammenpasst". Man kann sich das auch so merken: *Ein C-Compiler hat kein Gedächtnis.*

Während er *eine* Übersetzungseinheit bearbeitet "erinnert er sich nicht" an irgendwelche *anderen* Einheiten, die er vor kurzem bearbeitet hat.

Besonders diese Regel unterscheidet das klassische Übersetzungs-und-Ausführungsmodell von neueren Modellen. Wenn z.B. ein Java-Compiler eine Java-Quelldatei bearbeitet, öffnet er in der Regel weitere Dateien um zu prüfen, ob die aktuelle Quelldatei "mit den anderen Dateien zusammenpasst". D.h. ein Java-Compiler *hat* ein Gedächtnis.

Weil ein C-Compiler kein Gedächtnis hat, entsteht folgendes Problem:

Wenn man eine Funktion namens `fu10` in *einer* Implementierungsdatei `q10.c` definiert und in einer *anderen* Implementierungsdatei `q20.c` aufruft, kann der Compiler nicht ohne weiteres prüfen, ob der Aufruf und die Definition zusamenpassen. Selbst wenn man zuerst die Übersetzungseinheit `ÜE-q10` und danach `ÜE-q20` compiliert, weiß der Compiler bei der Bearbeitung von `ÜE-q20` nicht mehr, ob überhaupt eine Funktion namens `fu10` definiert wurde (auch wenn er die Definition vor wenigen Nanosekunden selbst gesehen hat) und kann sich schon gar nicht an an den Rückgabetyp und die Anzahl und die Typen der Parameter erinnern.

Aus diesem Grund muss der Programmierer dem Compiler in jeder Quelldatei, in der eine Funktion namens `fu10` aufgerufen aber nicht definiert wird, *versprechen*, in irgendeiner Quelldatei des Programms eine Funktion namens `fu10` mit einem bestimmten Rückgabetyp und bestimmen Parametern zu definieren. Ein solches Versprechen bezeichnet man offiziell als eine *Deklaration der Funktion* `fu10`. Solche Deklarationen können z.B. so aussehen:

Beispiel-03: Deklarationen von 2 Funktionen namens `fu10`

```
3 int  fu10(int n1, int n2);
4 void fu10(int * a1, int * a2, bool aufrunden);
```

Mit diesen Deklarationen verspricht der Programmierer dem Compiler, in irgendwelchen Quelldateien des Programms zwei Funktionen namens `fu10` mit den angegebenen Ergebnistypen und Parametern zu definieren.

Ein C-Compiler ist eher gutgläubig als mißtrauisch. Wenn er Versprechungen wie die in Zeile 3 und 4 in einer Übersetzungseinheit `ÜE-q10` sieht, vertraut er ihnen, bis er mit der Bearbeitung der Einheit fertig ist (dann vergisst er sie wieder). Von jedem Aufruf einer Funktion namens `fu10` in `ÜE-q10` prüft er, ob der Aufruf zu einer der versprochenen Funktionen passt. Falls nicht, meldet er einen entsprechenden Fehler.

Wenn der Compiler in einer Übersetzungseinheit einen Aufruf einer Funktion namens `fu20` sieht, ohne vorher eine entsprechende Definition oder wenigstens eine Deklaration gesehen zu haben, meldet er ebenfalls einen Fehler. Das gilt auch dann, wenn wenige Zeilen unterhalb des Aufrufs eine Deklaration oder sogar die Definition der Funktion steht.

Ganz ähnliche Regeln wie für *Funktionen* gelten auch für *Variablen*: Wenn man in einer Quelldatei `q10.c` eine Variable `v` benutzen will, die in einer anderen Quelldatei `q20.c` definiert wurde, muss man `v` in `q10.c` *deklarieren* (d.h. man muss dem Compiler versprechen, die Variable `v` in irgendeiner Quelldatei des Programms zu definieren). Erst in den Zeilen *nach* der Deklaration darf man die Variable dann benutzen.

Beispiel-03: Eine Variable definieren und deklarieren

Die Definition einer Variablen namens `v` in einer Quelldatei `q10.c`:

```
5 // Datei q10.c:
6 int v = 17;   // Auf Deutsch: Erzeuge eine Variable namens v vom
7              // Typ int mit dem Anfangswert 17.
```

Eine Deklaration der Variablen `v` in einer Quelldatei `q20.c`:

```
 8 // Datei q20.c:
 9 extern int v; // Auf Deutsch: Ich verspreche, eine int-Variable namens
10              // v in irgendeiner Datei dieses Programms zu definieren!
11
12 int w = 2*v;  // Hier wird v benutzt
13
14 int main() {
15    v = v+1;   // Hier wird v zweimal benutzt
16    ...
17 }
```

Obwohl der *C-Compiler* nur jeweils *eine* Übersetzungseinheit auf einmal bearbeitet (prüft und evtl. übersetzt), kann man dem `gcc` *beliebig viele* Implementierungsdateien angeben. Der `gcc` ruft dann entsprechend häufig den C-Compiler auf.

Beispiel-04: Mehrere Dateien mit einem Kommando compilieren lassen. Die `gcc`-Option `-c`

```
18 > gcc -c q10.c q20.c q30.c
```

Die Option `-c` bewirkt, dass die angegebenen Implementierungsdateien (`.c`-Dateien) nur compiliert werden (und nicht auch noch *gebunden* werden, siehe unten).

6. Objektdateien und Bibliotheken

Der C-Compiler (zusammen mit seinem kleinen Helfer, dem Präprozessor) übersetzt *Implementierungsdateien* (`.c`-Dateien) in *Objektdateien* (`.o`-Dateien). Eine Objektdatei ist im Wesentlichen eine Zusammenfassung von *Definitionen*. Mit dem Programm namens `nm` kann man sich die Namen der in einer Objektdatei definierten Größen anzeigen lassen.

Beispiel-01: Objektdateien mit `nm` untersuchen. Die `gcc`-Option `-g` , die `nm`-Option `-l` (klein ELL)

```
 1 > gcc -c q10.c    # q10.o erzeugen
 2 > gcc -c -g q20.c # q20.o mit Debug-Informationen darin erzeugen
 3 >
 4 > nm q10.o
 5 00000000 D anna1
 6 00000000 T plus1
 7 >
 8 > nm -l q20.o
 9 00000000 D bert2 /home/Karl/Bibliothek/d20.c:8
10 00000000 T mal2  /home/Karl/Bibliothek/d20.c:10
```

In der Objektdatei `q10.o` werden zwei Größen namens `anna1` und `plus2` definiert (Zeile 5 und 6). In der Objektdatei `q20.o` werden zwei Größen namens `bert2` und `mal2` definiert (Zeile 9 und 10). Da `q20.o` auch Debug-Informationen enthält (Option `-g` in Zeile 2) kann man sich von `nm` mit der Option `-l` (wie line nr) auch die *Zeilen-Nummern* anzeigen lassen, in denen die Definitionen von `bert2` bzw. `mal2` (in der Quelldatei `q20.c`) beginnen.

Objektdateien kann man in sogenannten *Bibliotheken* zusammenfassen. Das vereinfacht ihre Wiederverwendung in mehreren Programmen und beschleunigt den Zugriff auf sie.

Beispiel-02: Mit dem Archivprogramm `ar` drei Objektdateien in eine Bibliothek einfügen

```
11 > ar r uv libMeineBib.a d10.o d20.o d30.o
```

Die drei Objektdateien `d10.o`, `d20.o` und `d30.o` werden in die Bibliothek `libMeineBib.a` einfgefügt. Falls noch keine Datei namens `libMeineBib.a` existiert, wird vorher eine neue, leere Bibliothek mit diesem Namen erzeugt. Genauer:

Das `r` (wie "replace") in Zeile 11 bedeutet: Falls die Bibliothek schon exisitiert und eine Datei namens `d10.o` enthält, soll sie durch die angegebene Datei `d10.o` *ersetzt* werden. Entsprechend für die anderen Dateien.

Das u bedeutet: Eine Ersetzung soll nur stattfinden, wenn die angegebene Datei *neuer* ist als die Datei in der Bibliothek.

Das v ("verbose", geschwätzig) bedeutet auch hier: Bei der Ausführung des ar-Kommandos sollen möglichst viele informative Meldungen ausgegeben werden.

Anmerkung: Mit dem Programm ar (wie "archiver") kann man (ähnlich wie mit dem verbreiteten Programm zip) *beliebige* Dateien komprimieren und zu einem Archiv zusammenfassen. Eine *Bibliothek* ist ein ar-Archiv, welches nur *Objektdateien* enthält.

Beispiel-03: Den Inhalt einer Bibliothek mit dem Archivprogramm ar anzeigen lassen

```
12 > ar t libMeineBib.a
13 > ar tv libMeineBib.a
```

Das Kommando in Zeile 12 gibt die Namen aller Dateien aus, die sich im Archiv libMeineBib.a befinden. Das Kommando in Zeile 13 ist die geschwätzige Variante (kostet aber keine Fernsehgebühren).

Beispiel-04: Den Inhalt einer Bibliothek mit dem Programm nm anzeigen lassen

```
14 > nm meineBib.a
15 d1.o:
16 00000000 D anna1
17 00000000 T plus1
18
19 d2.o:
20 00000000 D bert2
21 00000000 T mal2
22
23 d3.o:
24 00000000 D langeRede
```

Das Programm nm gibt einem also nicht nur die Namen der *Objektdateien*, sondern für jede Objektdatei auch die Namen der darin *definierten Größen* (Funktionen und Variablen) aus.

Im Prinzip kann man den *Namen* einer Bibliothek *frei wählen*. Es ist aber besonders praktisch und empfehlenswert, den Namen mit dem Vorspann lib beginnen und mit der Erweiterung .a enden zu lassen (siehe unten).

7. Der Binder prüft Objektdateien und bindet sie zu einer ausführbaren Datei

Der Binder erledigt 2 Aufgaben:

1. Er prüft, ob alle Objektdateien eines Programms vollständig sind und zusammenpassen.
2. Er bindet alle Objektdateien eines Programms zu einer ausführbaren Datei zusammen.

Wenn man den Binder aufruft (meist tut man das indirekt über den gcc) muss man ihm im Wesentlichen eine oder mehrere *Objektdatei(en)* angeben, die er zusammenbinden soll.

Beispiel-01: Den Binder drei Objektdateien zu einer ausführbaren Datei zusammenbinden lassen

```
 1 > gcc d10.o d20.o d30.o -o prog.exe
```

Mit diesem Kommando befiehlt man dem Binder, die drei angegebenen Objektdateien zu einer Ausführbaren Datei namens prog.exe zusammenzubinden. Ein Compiler wird in diesem Beispiel nicht aufgerufen, weil alle angegebenen Dateien bereits Objektdateien (.o-Dateien) sind.

Anmerkung: Objektdateien sind grundsätzlich *nicht ausführbar*. Auch wenn ein Programm nur aus einer einzigen Objektdatei besteht, muss diese Objektdatei zu einer ausführbaren Datei "zusammengebunden werden".

Für jede der angegebenen Objektdateien d ermittelt der Binder zwei Mengen von Größen (d.h. von Funktionen und Variablen):

1. Größen die in `d` *definiert* werden.
2. Größen, die in `d` *benutzt* (aber nicht definiert) werden.

Anhand dieser Mengen prüft der Binder dann, ob jede (irgendwo) benutzte Größe (irgendwo) genau *einmal* definiert wird. Falls eine benutzte Größe *mehr als einmal* definiert wird, ist das auf jeden Fall ein Fehler und führt zum Abbruch des Bindevorgangs mit entsprechenden Fehlermeldungen.

Falls eine benutzte Größe in *keiner* der angegebenen Objektdateien definiert wird, führt das *nicht* automatisch zum Abbruch des Bindevorgangs. Der Binder hat vielmehr eine Liste von *Bibliotheken*. Wenn ihm eine Definition fehlt, sucht er erstmal in diesen Bibliotheken nach einer Objektdatei, die eine passende Definition enthält. Nur wenn auch diese Suche erfolglos bleibt, liegt ein Fehler vor und der Bindevorgang wird mit entsprechenden Fehlermeldungen abgebrochen.

Normalerweise gilt: Eine Kopfdatei enthält nur *Deklarationen* von Größen (und Definitionen von Typen). Alle Größen, die in den *Standardkopfdateien* deklariert werden, werden in gewissen *Standardbibliotheken* definiert. Die Bibliotheksliste des Binders enthält normalerweise mindestens alle Standardbibliotheken. Außerdem kann der Programmierer weitere Bibliotheken in die Bibliotheksliste des Binders eintragen lassen. Wenn man den `gcc` aufruft, kann man solche Bibliotheken auf verschiedene Weisen angeben, wie die folgenden Beispiele zeigen sollen.

Beispiel-02: Eine Bibliothek namens `KarlHeinz.abc` angeben

```
2 >  gcc d10.o d20.o d30.o KarlHeinz.abc -o prog.exe
```

Mit diesem Kommando befiehlt man dem Binder, die Objektdateien `d10.o`, `d20.o` und `d30.o` und falls nötig, weitere Objektdateien aus der angegebenen Bibliothek `KarlHeinz.abc` und aus den Standardbibliotheken zu einer ausführbaren Datei namens `prog.exe` zusammenzubinden. Wenn man auf diese Weise in einem `gcc`-Aufruf Bibliotheken festlegt, muss man ihren vollen Pfadnamen angeben. Der Aufruf im Beispiel-02 setzt also voraus, dass die Bibliothek `KarlHeinz.abc` im aktuellen Arbeitsverzeichnis steht. In größeren Projekten liegen Bibliotheken normalerweise in speziellen Verzeichnissen, in die man keine anderen Dateien schreiben darf.

Beispiel-03: Eine Bibliothek namens `libMeineBib.a` angeben, die `gcc`-Optionen `-L` und `-l`

```
3 >  gcc d10.o d20.o d30.o -L/projekt/bibs -lMeineBib -o prog.exe
```

Mit der Option `-L` kann man ein *Verzeichnis* angeben, in dem nach Bibliotheken gesucht werden soll. Im Beispiel wird das Verzeichnis `/projekt/bibs` angegeben.

Mit der Option `-l` (das soll ein kleines ELL sein) kann man den Namen einer *Bibliothek* angeben. Dabei muss man allerdings die folgende Regel beachten:

Regel für die Angabe von Bibliotheksnamen: Der tatsächliche Name der Bibliothek muss mit dem Vorspann `lib` beginnen und mit der Erweiterung `.a` enden, z.B. so: `libMeineBib.a`. Nach `-l` muss man den Namen aber *ohne* den Vorspann `lib` und *ohne* die Erweiterung `.a` angeben (sonst könnte ja jeder C-Programme binden lassen :-).

Im Beispiel-03 (Zeile 3) wird nach `-l` also nicht eine Bibliothek namens `MeineBib`, sondern die Bibliothek namens `libMeineBib.a` angegeben. Eine Bibliothek, deren Name nicht mit dem Vorspann `lib` beginnt oder nicht mit der Erweiterung `.a` endet, kann man auf diese Weise gar nicht angeben.

In einem Aufruf des `gcc` darf man die Optionen `-L` und `-l` beliebig oft verwenden und so beliebig viele Verzeichnisse und Bibliotheken angeben.

Anmerkung: Die Reihenfolge, in der die Bibliotheken in der Bibliotheksliste des Binders stehen, ist manchmal wichtig. Wenn es für eine bestimmte Größe *mehrere* Definitionen gibt wird *die* genommen, die der Binder zuerst findet.

Anmerkung: Wenn eine Bibliothek B (eine Objektdatei OB und darin) eine Funktion FB enthält, die eine Funktion FA aus (einer Objektdatei OA in) einer Bibliothek A aufruft, dann ist die Bibliothek B

abhängig von der Bibliothek A. Bibliotheken sollten möglichst unabhängig voneinander sein. Aber wenn eine Bibliothek B von einer Bibliothek A abhängig ist, muss man *zuerst* B und dann A angeben (obwohl die umgekehrte Reihenfolge vielen Menschen viel plausibler vorkommt).

Anmerkung: Auch wenn er nur eine einzige Definition aus einer (großen) Objektdatei benötigt, bindet der Binder immer die *ganze Objektdatei* ein. Deshalb sollten Objektdateien nicht zu groß sein und möglichst nur solche Größen (Funktionen und Variablen) enthalten, die typischerweise gemeinsam benutzt werden.

Beispiel-04: Angenommen, die Kopfdateien `K11.h`, `K12.h`, `k21.h`, `k22.h`, die Implementierungsdateien `d10.c`, `d20.c`, `d30.c` und die Bibliotheken `libA.a`, `libB.a` liegen alle im aktuellen Arbeitsverzeichnis. Dann kann man mit dem folgenden Kommando daraus eine ausführbare Datei namens `prog.exe` erzeugen lassen:

```
4 > gcc d10.c d120.c d30.c -L. -lA -lB -o prog.exe
```

Dieses Kommando veranlasst den `gcc`, dreimal den C-Compiler und dann einmal den Binder aufzurufen. Die Option `-L.` bewirkt, dass der Binder auch im aktuellen Arbeitsverzeichnis `.` nach den angegebenen Bibliotheken sucht.

Beispiel-05: Angenommen, die Kopfdateien `K11.h`, `K12.h`, `k21.h`, `k22.h` befinden sich im Verzeichnis `/prj/kdv`, die Implementierungsdateien `d10.c`, `d20.c`, `d30.c` befinden sich im aktuellen Arbeitsverzeichnis und die Bibliotheken `libA.a`, `libB.a`. `libC.a` befinden sich im Verzeichnis `/prj/lib`. Dann kann man mit dem folgenden Kommando daraus eine ausführbare Datei namens `prog.exe` erzeugen lassen:

```
5 > gcc -I/prj/kd d10.c d120.c d30.c -L/prj/lib -la -lB -o prog.exe
```

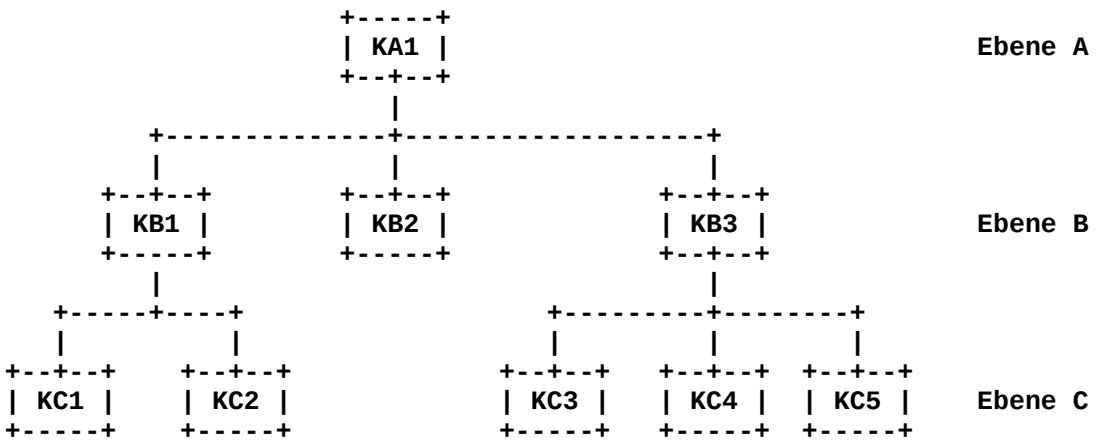
Auch dieses Kommando veranlaßt den `gcc`, dreimal den C-Compiler und dann einmal den Binder aufzurufen. Die Option `-I/prj/kdv` bewirkt, dass der Präprozessor Kopfdateien auch im Verzeichnis `/prj/kdv` sucht. Die Option `-L/prj/lib` bewirkt, dass der Binder auch im Verzeichnis `/prj/lib` nach den drei angegebenen Bibliotheken sucht.

8. Dokumentationen mit info lesen

Zu vielen quelloffenen Programmen gibt es Dokumentation in Form sogenannter *info-Dateien*, die man mit dem *info-Kommando* lesen kann. Z.B. sind die Programme `gcc`, `ar` und `nm` auf diese Weise dokumentiert worden. `Info`-Dateien können (ähnlich wie HTML-Dateien) *Links* enthalten, die auf Stellen innerhalb derselben Datei oder in anderen `info`-Dateien zeigen. Das `info`-Kommando ist eine Art Browser, den man allerdings nur mit der *Tastatur* bedienen kann (auf *Mauskommandos* reagiert er nicht). Es folgt hier eine Kurzanleitung für das Navigieren mit dem `info`-Browser.

Eine Info-Datei enthält *Knoten* (engl. nodes). Diese Knoten sind so miteinander verlinkt, dass sie einen *Baum* bilden. D.h. ganz oben gibt es einen *Wurzelknoten* (top), jeder andere Knoten "hängt an" einem *Mutterknoten* und an jedem Knoten können beliebig viele (0 oder mehr) *Tochterknoten* hängen. Jeder Knoten enthält Textzeilen (bis zu ein paar hundert) mit Informationen.

Beispiel-01: Die Baumstruktur einer `info`-Datei



In diesem Beispiel ist `KB1` der *Mutterknoten* von `KC1` und `KC2` und eine *Tochter* von `KA1`.

1. Starten des info-Browsers (Beispiele):

```
1 > info gcc
2 > info info
3 > info
```

In Zeile 1 wird der oberste Knoten mit Informationen zum `gcc` angezeigt. Mit dem 2. Kommando bekommt man Informationen zum `info`-Brauser. Das 3. Kommando öffnet den allerobersten `info`-Knoten (an dem alle anderen `info`-Knoten direkt oder indirekt dranhängen).

2. Beenden des info-Browsers: Durch Eingabe von `q` (wie "quit").

3. Innerhalb eines Knotens navigieren: Mit den Tasten `Pfeil-rauf`, `Pfeil-runter` (eine Zeile), `Bild-rauf` und `Bild-runter` (mehrere Zeilen). Mit `b` zum Beginn des Knotens.

4. Zwischen Töchtern einer Mutter navigieren: Mit `n` zur nächsten Tochter derselben Mutter, mit `p` (wie "previous") zur vorigen Tochter derselben Mutter (z.B. mit `n` von `KB1` zu `KB2` oder von `KC4` zu `KC5` etc., mit `p` von `KB2` zu `KB1` oder von `KC4` zu `KC3` etc.). Zur jeweils *ersten* Tochter einer Mutter (z.B. `KB1`, `KC3`) gibt es *keine vorige* Tochter, zur jeweils *letzten* Tochter (z.B. `KB3`, `KC2`) gibt es *keine nächste* Tochter. Der oberste Knoten einer `info`-Datei (im Beispiel: `KA1`) hat seine erste Tochter (im Beispiel: `KB1`) als Nachfolger.

5. Zur Mutter des aktuellen Knotens ("eine Ebene rauf"): `u` (wie "up").

6. Zum obersten Knoten in der aktuellen Datei: `t` (wie "top")

7. Zurück zum vorigen Knoten (bei dem man zuletzt war): `l` (wie "last"). Kann auch mehrmals angewendet werden. Sehr nützlich, wenn man "sich verlaufen hat".

8. Über Knotengrenzen hinweg durch eine Datei navigieren: In einer `info`-Datei stehen nach jedem Knoten erstmal seine Töchter. Für das Beispiel-01 ergibt sich daraus die Reihenfolge:

`KA1`, `KB1`, `KC1`, `KC2`, `KB2`, `KB3`, `KC3`, `KC4`, `KC5`.

Durch diese Knotenfolge kann man sich schrittweise vorwärts und rückwärts bewegen, wobei ein *Schritt* etwa soviele Zeilen umfasst, wie in das verwendete Fenster passen. Die `Blank`-Taste (die große Taste zwischen `Alt` und `Alt Gr`) bringt einen um einen Schritt *vorwärts*, die Rückwärts-Taste (die normalerweise das zuletzt eingegebene Zeichen löscht) geht einen Schritt *zurück*. Indem man wie-

derholt auf die Blank-Taste drückt, kann man also im Beispiel-01 vom Knoten KA1 bis zum Knoten KC5 gelangen, und mit der Rückwärts-Taste von KC5 bis KA1.

9. Suchen (nach Begriffen, die im Index stehen): i (wie "index") eingeben, dann den zu suchenden Begriff und Return. Weitersuchen mit einem Komma , .

10. Suchen (nach einer beliebigen Zeichenkette): s (wie "search") eingeben, dann die zu suchende Zeichenkette und Return. Weitersuchen mit s Return.

11. Ein Übersicht über alle Tastenkommandos anzeigen lassen: ? eingeben. Ein separates Fenster mit den Kommandos darin geht auf. Mit Strg-x 0 kann man das Fenster wieder schließen. In der Übersicht bezeichnet der Buchstabe C (wie "control") die Strg-Taste, RET die Return-Taste.

12. Einem Link folgen: Ein Link beginnt mit einem Stern * und endet mit einem oder zwei Doppelpunkten :: (z.B. * Directory node: oder * Portability::). Wenn irgendwo in der aktuellen Zeile (in der der Cursor steht) ein Link vorkommt und man Return drückt, springt man zu der durch den Link bezeichneten Stelle. Mit l (klein ELL) kann man wieder zurückspringen.

9. Anhang A: Standard-Kopfdateien

Jeder mit dem Standard C99 konforme C-Ausführer muss die folgenden 25 Kopfdateien kennen:

assert.h	Testbefehl assert
complex.h	Komplexe Arithmetic
ctype.h	Zeichen klassifizieren: isalpha, isupper, islower, isdigit, toupper, ...
errno.h	Fehlernummern verwalten (EDOM, EILSEQ, ERANGE)
fenv.h	Gleitpunktrechnungen hardwarenah kontrollieren
float.h	Gleitpunkt-Konstanten (FLT_MAX, DBL_MAX, LDBL_MAX, FLT_MIN, ...)
inttypes.h	Standardisierte Ganzzahlttypen, Makros für printf (siehe auch stdint.h)
iso646.h	Namen and, bitand, bitor etc. für die Operationen &&, &, | etc.
limits.h	Ganzzahl-Konstanten (INT_MAX, INT_MIN, UINT_MAX, SHRT_MAX, ...)
locale.h	Internationalisierung, Zahlen und Datumsangaben landspezifisch formatieren etc.
math.h	Funktionen (sin, cos, sqrt, pow, lround, ceil, floor, nan, nextafter, ...)
setjump.h	Weite Sprungbefehle, jmp_buf, setjmp, longjmp
signal.h	Behandlung von Signalen (ähnlich wie Ausnahmen)
stdarg.h	Funktionen mit einer variablen Anzahl von Parametern
stdbool.h	bool, true, false
stddef.h	Typen ptrdiff_t, size_t, wchar_t, Makro NULL
stdint.h	Standardisierte Ganzzahlttypen int_least8_t, int_least16_t, ...
stdio.h	Bearbeiten von Dateien, Typen FILE, fpos_t
stdlib.h	Funktionen abs, atof, atoi, atol, bsearch, calloc, div, exit, free, getenv, malloc, NULL, qsort, rand, size_t, srand, strtof, strtoi, ...
string.h	Funktionen strcpy, strncpy, strcat, strncat, strcmp, strncmp, ...
tgmath.h	Generische Gleitpunktrechnung (float, double, long double leicht änderbar)
time.h	Datum und Zeit, Typen clock_t, time_t, Funktionen clock, difftime, ...
wchar.h	Multibyte-Zeichen, Funktionen fwprintf, fwscanf, ...
wctype.h	Multibyte-Zeichen klassifizieren: iswalpha, iswdigit, towupper, ...

10. Anhang B: Übersicht über die hier behandelten gcc-Optionen

Option	Erläuterung
-c	Nur (präprozessieren und) compilieren, nicht binden
-C	Der Präprozessor soll Commentare aus den Quelldateien (`.c`- und `.h`-Dateien) in die Übersetzungseinheit übernehmen. Nur zusammen mit -E sinnvoll
-E	Nur präprozessieren, nicht compilieren (und schon gar nicht binden)
-g	Der Compiler soll Debug-Informationen in die Objektdatei schreiben (z.B. Zeilen-Nrn)
-I	Verzeichnis, in dem der Präprozessor nach Kopfdateien (`.h`-Dateien) suchen soll
-L	Verzeichnis, in dem der Binder nach Bibliotheken suchen soll
-l	Name einer Bibliothek, in der der Binder nach fehlenden Definitionen suchen soll
-o	Name der Ausgabedatei (z.B. einer Objektdatei oder einer ausführbaren Datei) festlegen
-M	Die Namen aller Kopfdateien (`.h`-Dateien) ausgeben, die von einer Implementierungsdatei (`.c`-Datei) inkludiert werden (eigentlich zum Erzeugen von `make`-Dateien gedacht)
-MM	Wie -M, es werden aber keine Namen von Standard-Kopfdateien ausgegeben
-v	Möglichst viele Meldungen ausgeben