Die Programmiersprache C

von Ulrich Grude

Unterlagen für eine Lehrveranstaltung im Fach Programmierung 1 (PR1-TB1)
im WS06/07 im Fachbereich Informatik der TFH Berlin, Version 0.8
Verbesserungsvorschläge und Hinweise auf Fehler sind jederzeit willkommen
e-mail: grude@tfh-berlin.de

Inhaltsverzeichnis

1 Literatur......3

2 Der Geist von C......6

3 Globale Struktur eines C-Programms......6

4 Vereinbarungen......6

5 Zwei Hallo-Programme......7

6 Compilieren und Binden, Definitionen und Deklarationen......8

7 Funktionsdefinitionen und lokale ("interne") Vereinbarungen......11

8 Der Präprozessor und der Compiler......12

9 Kopfdateien (header files)......12

10 Der Binder......17

11 Sichtbarkeit......18

12 Vorvereinbarte und vom Programmierer vereinbarte Typen......21

13 Aufzählungstypen (enum types)......22

14 Verbundtypen (struct types)......23

15 Adressen, der Adressoperator & und der Zieloperator *......25

16 Reihungen (arrays) und Adressen......30

17 Den Cursor positionieren und Zeichen ohne Return einlesen......34

18 Anhang: Compilieren und Binden ohne und mit make......39

19 Anhang: Die Compiler gcc und lcc......43

20 Anhang: Kurzübersicht über Standard-Kopfdateien......45

21 Anhang: Der Editor vi......46

22 Anhang: Der Editor vim und mehr vi(m)-Befehle......48

Die Programmiersprache C

1 Literatur

Brian W. Kernighan, Dennis M. Ritchie
"The C Programming Language", Second Edition
Prentice Hall 1988, 272 Seiten, ca. 45,- €

Das Buch von den "Erfindern von C", sehr lesbar geschrieben und übersichtlich aufgebaut, mit vielen kleinen, sehr sorgfältig ausgewählten Beispielen und vielen kleinen und größeren Übungsaufgaben.. Enthält allerdings kaum Kritik an C und beruht auf dem veralteten ANSI-Standard (Programming Language C, X3. 159-1989). Trotzdem sehr empfehlenswert. Es gibt auch eine Übersetzung ins Deutsche (erschienen im Hanser Verlag).

C. L. Tondo, S. E. Gimpel
"The C Answer Book, Second Edition"
Prentice Hall 1989, ca. 130 Seiten, ca. 15,- €

Das C-Buch von Kernighan und Ritchie enthält ungefähr 100 Übungsaufgaben. Dieses Buch enthält Lösungen zu diesen Aufgaben. Es gibt auch eine Übersetzung ins Deutsche (erschienen im Hanser Verlag).

"Programming Languages -- C, International Standard ISO/IEC 9899:1999 (E)"
(in Englisch, keine deutsche Version verfügbar)
siehe z.B. www2.beuth.de oder www.iso.org (webstore)
554 Seiten, ca. 276,- €! Als Datei (im .pdf-Format) nur ca 20,- €.

Das ist der neuste und verbindliche Standard der Sprache C (Kurzname: "C99"). Zur Zeit werden viele C-Implementierungen auf diesen Standard umgestellt und realisieren schon Teile davon (kaum eine realisiert den vollständigen Standard). Sehr genau formuliert, aber sehr kompliziert und schwer zu lesen. Pflichtlektüre wenn man einen C-Compiler programmieren will.

"Rationale for International Standard -- Programming Languages -- C"
Im Internet (auf Englisch) in verschiedenen Formaten (PDF, HTML, ...) kostenlos verfügbar, als .pdf-Datei z.B. unter folgender Adresse:
`www.dkuug.dk/JTC1/SC22/WG14/www/docs/n897.pdf`

Als ***rationale*** bezeichnet man im Englischen ein Papier mit Begründungen und Erläuterungen (z.B. zum Entwurf einer Programmiersprache). Dieses Papier hilft einem also zu verstehen, warum C bestimmte Eigenschaften hat und warum die Entwickler von C bestimmte Probleme so und nicht anders gelöst haben. Sehr interessante Lektüre und besonders hilfreich, wenn man sich über irgend ein Detail der Sprache ärgert und sich fragt, warum die Entwickler das nicht "besser geregelt" haben.

S. P. Harbison, G.L. Steele
"C: A Reference Manual", 5th Edition (!)
Prentice Hall (Pearson Education) 2002, 533 Seiten, ca. 47,- €.
(in Englisch, keine deutsche Version verfügbar)

Eine hervorragend klare und lesbare Beschreibung der wichtigsten C-Dialekte (Traditionelles C aus der Zeit vor 1990, Standard C von 1989, Standard C von 1999 und C++-kompatibles C), einschließlich der (Standard-) Bibliotheken dieser Dialekte. Dieses Buch macht keine Werbung für C, sondern stellt die Sprache möglichst sachlich dar, mit all ihren Stärken und erheblichen Problemen. Gute Fachbücher erkennt man an der hohen Ausgaben-Nummer (3. Ausgabe, 4. Ausgabe, ...). Einen guten C-

Programmierer erkennt man u.a. an diesem Buch auf seinem Schreibtisch (ein sehr guter C-Programmierer hat es im Kopf).

H. Erlenkötter
"C Bibliotheksfunktionen sicher anwenden"
rororo 2003, 415 Seiten, ca. 13,- €.

Eine gut lesbare Sammlung von Beispielen, die zeilenweise erläutert werden, weniger als Nachschlagewerk geeignet. Berücksichtigt teilweise den C99-Standard. Preiswert.

Helmut Herold
"C-Kompaktreferenz"
Addison-Wesley (Pearson Education) 2003, 463 Seiten, ca. 20,- €.

Mischung aus Lehrbuch und Referenz. Enthält auch (auf etwa 100 Seiten) einige wichtige Algorithmen (Quick-Sort, binäres Suchen, Permutationen, Baumalgorithmen, Quadratwurzel Newton-Iteration, ... etc.). Informationen zum C99-Standard fehlen leider. Für ältere Leser ist die Schrift zu klein.

J. Goll, U. Bröckl, M. Dausmann
"**C als erste Programmiersprache**", 4. Auflage
Teubner 2003, 554 Seiten, ca. 25,- €.
Ein didaktisch ordentlich aufgebautes und umfangreiches Lehrbuch (mit CD). An einigen Stellen ein bisschen umständlich. Noch kein Hinweis auf den C99-Standard. Preis/Leistung sehr gut.

Karlheinz Zeiner
"**Programmieren lernen mit C**", 4. aktualisierte und überarbeitete Auflage
Carl Hanser Verlag 2000, 360 Seiten, ca. 25,00 €

Lesbar geschrieben und übersichtlich aufgebaut. Enthält auch einen Ausblick auf dem C99-Standard.

Alan R. Feuer
"C-Puzzlebuch"
Carl Hanser Verlag 1980, 190 Seiten, ca. 25,00 € ?

Enthält zahlreiche Beispiele für schwer lesbare und unlesbare Teile von C-Programmen (in Form von "Denksportaufgaben"). Unterhaltsam, nützlich, manchmal bizarr. Zeigt die Sprache C aus einer ungewöhnlichen Perspektive.

Jürgen Schönwälder
"Programmieren in C"
Dieses Skript (als .ps-Datei), Programme und Übungsunterlagen findet man unter der Netzadresse
http://www.vorlesungen.uni-osnabrueck.de/informatik/cc02/

A. Oram, S. Talbott
"**Managing Projects With make**", 2nd Edition
O'Reilly 1993, 150 Seiten, ca 25,- € ?

Sobald ein C-Programm aus mehreren Dateien besteht, sollte man das Compilieren und Binden (und evtl. weitere Schritte wie das Testen und Installieren des Programms, das Löschen alter Versionen etc.) in einer **Makedatei** beschreiben und vom Programm **make** durchführen lassen. Dieses Programm ist ein ziemlich allgemeiner und mächtiger Kommandointerpreter (Kenner sollen damit sogar Kaffee kochen können) und jeder C-Programmierer muss es zumindest in seinen Grundzügen kennen. Das Buch von Oram und Talbott ist eine Art make-Referenz. Es ist sehr lesbar geschrieben (mit vielen kleinen Beispielen) und behandelt auch die tückischen Unterschiede zwischen Varianten von `make`.

S. Oualline
"**Vi IMproved - Vim**", (für vim Version 5.7)
New Riders Publishing 2001, 572 Seiten, ca. 53,- €

Eine vollständige, systematische und sehr verständliche Beschreibung des Editors vim von einem sehr kompetenten Kenner. Sehr empfehlenswert für ernsthafte und intensive Benutzer des vim.

L. Lamb, A. Robbins
"**Learning the vi Editor**", 6th Edition
O'Reilly 1998, 327 Seiten, ca. 30,- €

Eine "sanfte" Einführung in die unüberschaubar vielen Befehle des Editors **vi** und seiner wichtigsten Varianten (**vim** von Bram Moolenaar, **elvis** von Steve Kirkendall, **nvi** von Keith Bostic und **vile** von Kevin Buettner, Tom Dickey und Paul Fox). Die einfachen vi-Befehle werden einfach erklärt, aber einige kompliziertere Befehle werden nur kurz oder nicht behandelt.

C. Newham, B. Rosenblatt
"**Learning the bash Shell**", 2nd Edition (für bash Version 2.0)
O'Reilly1998, 318 Seiten, ca. 30,- €

Der Kommandozeileninterpreter **bash** steht unter Linux, unter anderen Unix-Varianten und unter verschiedenen Windows-Varianten (95/98/me/2000/XP) zur Verfügung. Die Sprache C hat sich mit dem Betriebssystem Unix verbreitet und Kommandozeileninterpreter wie **bash** sind ein wichtiger Teil der "Unix- und C-Kultur".

2 Der Geist von C

Das folgende Zitat stammt aus dem Papier "Rationale for International Standard Programming Language C, Revision 2, 20 October 1999":

Some of the facets of the spirit of C can be summarized in phrases like

- **Trust the programmer.**
- **Don't prevent the programmer from doing what needs to be done.**
- **Keep the language small and simple.**
- **Provide only one way to do an operation.**
- **Make it fast, even if it is not guaranteed to be portable.**

Ende des Zitats.

3 Globale Struktur eines C-Programms

Ein C-Programm besteht aus **Quelldateien** (beliebig vielen, mindestens einer).

4 Vereinbarungen

Jede Quelldatei enthält eine Reihe von **externen Vereinbarungen**.

Beispiel 1: Struktur eines C-Programms, schematisch

```
 1 // Datei 1
 2 Externe Vereinbarung 1
 3 Externe Vereinbarung 2
 4 Externe Vereinbarung 3
```

```
 5 // Datei 2
 6 Externe Vereinbarung 1
 7 Externe Vereinbarung 2
```

Vereinbaren kann bzw. muss man unter anderem folgende Größen:

- **Variablen** (veränderbare und unveränderbare)
- **Funktionen** (void-Funktionen und solche mit einem "richtigen" Rückgabetyp)
- **Typ-Bezeichner** (als Synonyme für schon vereinbarte Typ-Bezeichner)
- **Typen** (struct-, union- und enum-Typen, d.h. Verbund-, Vereinigungs- und Aufzählungstypen).

Beispiel 2: Vereinbarungen von Variablen

```
 8 int   anz;
 9 float f17 = 123.45;
```

Der Bezeichner `anz` soll eine Variable vom Typ `int` bezeichnen. Der Bezeichner `f17` soll eine Variable vom Typ `float` mit dem Anfangswert `123.45` bezeichnen.

Beispiel 3: Vereinbarungen von Funktionen

```
10 int summe(int, int);
11
12 int hoch(int basis, int exponent) {
13    anz++;
14    int erg = 1;
15    while (exponent > 0) {
16       erg *= basis;
17       exponent--;
18    }
19    return erg;
20 } // hoch
```

Der Bezeichner `summe` soll eine Funktion mit zwei Parametern vom Typ `int` und dem Rückgabetyp `int` bezeichnen.

Der Bezeichner `hoch` soll ebenfalls eine Funktion mit zwei Parametern vom Typ `int` und dem Rückgabetyp `int` bezeichnen. Diese Vereinbarung legt auch den Rumpf der Funktion fest (Zeilen 13 bis 21). Der Rumpf beschreibt, wie das Ergebnis der Funktion berechnet werden soll und welchen Seiteneffekt die Funktion hat (sie verändert die Variable `anz`).

Beispiel 4: Vereinbarung eines Typ-Bezeichners und eines Typs

```
21 typedef int ganz;
22
23 typedef struct {
24    float x;
25    float y;
26    ganz  gewicht;
27 } punkt;
```

Der Bezeichner `ganz` soll ein Synonym für den Bezeichner `int` sein, d.h. der Bezeichner `ganz` soll (auch) den Typ `int` bezeichnen. Der Bezeichner `punkt` soll einen Verbundtyp (`struct type`) bezeichnen. Jeder Wert dieses Typs soll drei Komponenten enthalten, zwei vom Typ `float` und eine vom Typ `ganz` (d.h. vom Typ `int`).

Ein C-Programm kann aus beliebig vielen externen Vereinbarungen bestehen, muss aber mindestens **eine** externe Vereinbarung enthalten, nämlich die Vereinbarung einer Funktion mit dem Bezeichner `main`.

Die **Ausführung** eines C-Programms besteht aus den folgenden beiden Schritten:
Schritt 1: Alle externen Vereinbarungen des Programms werden ausgeführt.
Schritt 2: Die Funktion `main` wird ausgeführt.

Anmerkung: Genau **eine** Quelldatei eines C-Programms muss die Vereinbarung der `main`-Funktion enthalten. In Java dürfen dagegen mehrere Klassen, die zum selben Programm gehören, `main`-Methoden enthalten.

5 Zwei Hallo-Programme

```
 1 // Datei hallo01.c
 2
 3 #include <stdio.h> // fuer printf
 4
 5 int main() {
 6    printf("------------------------------------------\n");
 7    printf("Hallo aus einem C-Programm namens hallo01!\n");
 8    printf("------------------------------------------\n");
 9
10    return 0; // Meldung an das Betriebssystem: Alles o.k.
11 } // main
12 /* ------------------------------------------------------------------------
13 Ausgabe des Programms Hallo01:
14
15 ------------------------------------------
16 Hallo aus einem C-Programm namens hallo01!
17 ------------------------------------------
18 ------------------------------------------------------------------------ */
```

```
19 // Datei hallo02.c
20
21 #include <stdio.h> // fuer scanf und printf
```

```
22 // ------------------------------------------------------------------------
23 int main() {
24    int   n1; // Aeltere C-Compiler erlauben Variablenvereinbarungen
25    float f1; // nur am Anfang einer Funktion, vor der ersten
26    float f2; // Anweisung (z.B. vor der ersten printf-Anweisung)
27
28    printf("------------------------------------------\n");
29    printf("Eine Ganzzahl? ");
30    scanf ("%d", &n1);          // Liest eine Ganzzahl nach n1
31    printf("Ihre Eingabe:  %d\n", n1);
32    printf("------------------------------------------\n");
33    printf("Zwei Gleitpunktzahlen? ");
34    scanf ("%f %f", &f1, &f2); // Liest zwei Bruchzahlen nach f1 und f2
35    printf("Ihre Eingaben:   %f, %f!\n", f1, f2);
36    printf("------------------------------------------\n");
37
38    return 0; // Meldung an das Betriebssystem: Alles o.k.
39 } // main
40 /* ------------------------------------------------------------------------
41 Ein Dialog mit dem Programm hallo02:
42
43 ------------------------------------------
44 Eine Ganzzahl? 123
45 Ihre Eingabe:  123
46 ------------------------------------------
47 Zwei Gleitpunktzahlen? 1.23 45.6
48 Ihre Eingaben:   1.230000, 45.599998!
49 ------------------------------------------
50 ---------------------------------------------------------------------- */
```

Aufgabe 1: Schreiben Sie ein C-Programm namens `hallo03`, welches wiederholt zwei Ganzzahlen `n1` und `n2` einliest (mit `scanf`) und ihr Produkt `n1 * n2` und ihren Quotienten `n1 / n2` ausgibt (mit `printf`), bis der Benutzer das Programm durch Eingabe von `<Strg>-c` abbricht.

6 Compilieren und Binden, Definitionen und Deklarationen

Die Sprache Java beruht auf einem relativ **modernen** Compilations-Modell und ein Java-Programmierer braucht sich in aller Regel nicht im Einzelnen darum zu kümmern, welche Aufgaben vom Java Compiler und welche vom Java-Interpreter (der virtuellen Maschine) erledigt werden. Es genügt, wenn er sich einen **abstrakten Java-Ausführer** vorstellt, dem er seine Programme übergibt und der diese Programme prüft und ausführt.

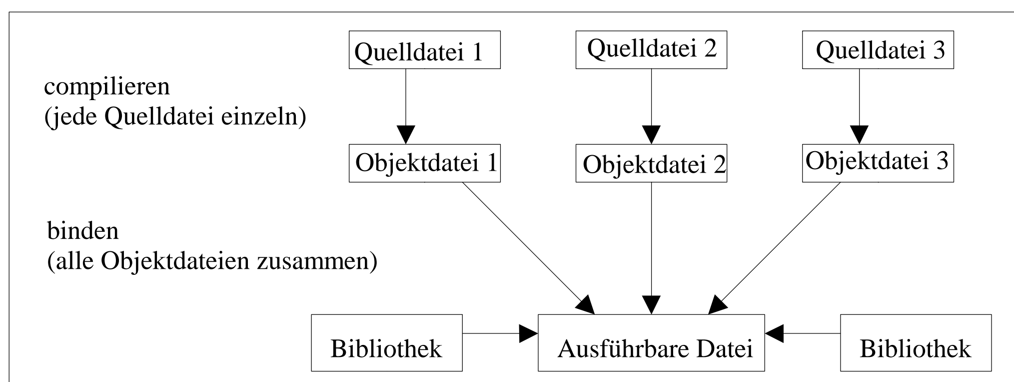
Die Sprache **C** beruht auf einem **relativ alten** Compilations-Modell und man kann zahlreiche Regeln der Sprache nur dann richtig verstehen, wenn man zumindest mit den Grundzügen dieses Modells vertraut ist.

Ein C-Ausführer besteht unter anderem aus einem **Compiler** und aus einem **Binder**. Der Compiler prüft jeweils **eine** Quelldatei und übersetzt sie (falls er keine formalen Fehler findet) in eine entsprechende **Objektdatei**. Der Binder prüft alle Objektdateien eines Programms noch einmal und bindet sie (falls er keine formalen Fehler findet) zu einer **ausführbaren Datei** zusammen.

Anmerkung 1: Der Begriff einer **Objektdatei** ist älter als die objektorientierte Programmierung und hat **nichts** mit ihr zu tun.

Anmerkung 2: Unter DOS und Windows bezeichnet man die **ausführbaren Dateien**, die ein Binder erzeugt, auch als **.exe-Dateien**. Unter einem Unix-Betriebssystem erkennt man ausführbare Dateien nicht an ihrem Namen, sondern an bestimmten "Ausführungsberechtigungsbits".

Die folgende Graphik soll anschaulich machen, wie aus mehreren **Quelldateien** eines Programms eine **ausführbare Datei** erzeugt wird:



Das **Grundproblem** dieses Compilations-Modells: Der Compiler prüft und übersetzt immer nur **eine Datei** auf einmal und "hat kein Gedächtnis". Das heißt: Wenn er eine Datei übersetzt, "weiß er nichts" von irgendwelchen anderen Dateien, die möglicherweise zum selben Programm gehören. Wird eine Variable oder Funktion z.B. in der Quelldatei 1 **definiert** und in der Quelldatei 2 **benutzt**, dann prüft der Compiler **nicht**, ob die Definition und die Benutzung "zusammenpassen". Erst der Binder prüft, ob alle **Objektdateien** eines Programms konsistent sind und ob jede Größe, die irgendwo **benutzt** wird, auch irgendwo **definiert** wird (oder ob sich eine geeignete Definition in einer Bibliothek des Binders befindet).

Damit trotzdem schon der **Compiler** ein paar wichtige Prüfungen durchführen kann, hat man in C zusätzlich zu den **eigentlichen** Vereinbarungsbefehlen (den **Definitionen**) die **Deklarationen** eingeführt, die dem Compiler **versprechen**, dass eine bestimmte Größe in irgendeiner Datei des Programms definiert wird. Die folgenden Regeln betreffen nur **VF-Größen**, d.h. Variablen und Funktionen. **Typbezeichner** und **Typen** werden weiter unten behandelt.

Regeln über Deklarationen und Definitionen:

DekDef 1: Man darf **jede VF-Größe** in **jeder Datei** eines Programms **beliebig oft deklarieren**.

DekDef 2: Man darf eine **VF-Größe** in höchstens **einer** Datei eines Programms einmal **definieren**. Man **muss** eine **VF-Größe** (nur dann) **definieren**, wenn sie irgendwo im Programm **benutzt** wird. Diese Regel ist auch unter dem Namen **one definition rule** (ODR) bekannt.

DekDef 3: Eine **VF-Größe** muss man in jeder Datei, in der sie benutzt wird, mindestens einmal **deklarieren** (oder voll definieren), und zwar **vor** der Zeile, in der sie zum ersten Mal benutzt wird.

DekDef 4: Alle Deklarationen einer **VF-Größe** innerhalb derselben Datei müssen "zusammenpassen" und dürfen sich nicht widersprechen.

Beispiel 1: Zusammenpassende Deklarationen

```
1 int summe(int otto, int emil);
2 ...
3   int           summe   (int,int);
4 ...
5 int summe(int anna, int berta) {return anna + berta;}
```

Eine Funktion namens `summe` wird in Zeile 1 und in Zeile 3 **deklariert** und in Zeile 5 **definiert** (und damit auch noch einmal deklariert). Die drei Vereinbarungen passen zusammen, da die Namen der Parameter und transparente Zeichen (engl. white space) keine Rolle spielen.

Allgemein unterscheidet man zwischen so genannten **externen Vereinbarungen** (die direkt in einer Datei stehen) und **lokalen Vereinbarungen** (die innerhalb einer Funktionsdefinition stehen).

Beispiel 2: Definitionen und Deklarationen

```
 1 // Datei sVereinbarungen01.c
 2 // ------------------------------------------------------------
 3 int   anz;                        // Externe Vereinbarung 01
 4 float f17 = 123.45;               // Externe Vereinbarung 02
 5
 6 int   summe(int, int);            // Externe Vereinbarung 03
 7 int   verdopple(int);             // Externe Vereinbarung 04
 8 // ------------------------------------------------------------
 9 int hoch(int basis, int exponent)  // Externe Vereinbarung 05
10 {
11    anz++;
12    int erg = 1;                    // Interne (lokale) Vereinbarung
13    while (exponent > 0) {
14       erg *= basis;
15       exponent--;
16    }
17    return erg;
18 } // hoch
19 // ------------------------------------------------------------
20 int main(void)                    // Externe Vereinbarung 06
21 {
22    int printf(const char *, ...);  // Interne (lokale) Vereinbarung
23
24    printf("Hallo!\n");
25    printf("verdopple(3): %3d\n", verdopple(3));
26    return 0;
27 } // main
```

```
28 // Datei sVereinbarungen02.c
29
30 extern int anz;                   // Externe Vereinbarung 01
31 int summe(int, int);              // Externe Vereinbarung 02
32 // ------------------------------------------------------------
33 int verdopple(int n) {            // Externe Vereinbarung 03
34    anz++;
35    return summe(n, n);
36 } // verdopple
37 // ------------------------------------------------------------
38 int summe (int n1, int n2) {      // Externe Vereinbarung 04
39    anz++;
40    return n1 + n2;
41 } // summe
```

In Zeile 3 wird eine Variable namens `anz` vereinbart. Ob es sich dabei um eine Deklaration oder Definition handelt, folgt nur aus dem Gesamtzusammenhang. Da sonst nirgends eine Variable namens `anz` definiert wird, gilt die Vereinbarung in Zeile 3 als **Definition**.

In Zeile 30 wird die Variable `anz` **deklariert**. Dass es sich hier um eine Deklaration handelt, folgt aus dem Schlüsselwort `extern`. Diese Deklaration ist ein **Versprechen** des **Programmierers** an den **Compiler**, dass in irgendeiner Datei dieses Programms eine `int`-Variable namens `anz` **definiert** wird. Der Compiler ist "gutgläubig" und vertraut dem Programmierer. Erst der Binder überprüft später, ob das Versprechen tatsächlich erfüllt wurde (allerdings nur, wenn die Variable `anz` irgendwo benutzt wird).

In Zeile 4 wird eine Variable namens `f17` **definiert**. Dass es sich um eine **Definition** handelt erkennt man an der Initialisierung (`= 123.45`).

In Zeile 6 und in Zeile 31 wird eine Funktion namens `summe` **deklariert** (ohne Rumpf). Diese Deklaration ist ein **Versprechen** des **Programmierers** an den **Compiler**, dass in irgendeiner Datei dieses Programms eine Funktion namens `summe` mit zwei `int`-Parametern und dem Rückgabetyp `int`

definiert wird. Der Compiler vertraut diesem Versprechen und der Binder überprüft später, ob es eingehalten wurde (allerdings nur, wenn die Funktion `summe` irgendwo benutzt wird).

Ab Zeile 38 wird die Funktion `summe` **definiert** (mit Rumpf).

Ab Zeile 9 wird eine Funktion namens `hoch` **definiert** (mit Rumpf).

Ab Zeile 20 wird eine Funktion namens `main` **definiert** (mit Rumpf). Diese Funktion hat null Parameter (das wird durch das Wort `void` in Zeile 19 ausgedrückt).

In Zeile 25 wird die Funktion `verdopple` aufgerufen. Das ist nur erlaubt, weil die Funktion vorher in derselben Datei vereinbart wurde (in Zeile 7 wurde sie deklariert).

In Zeile 35 wird die Funktion `summe` **aufgerufen**. Das ist nur deshalb erlaubt, weil die Funktion vorher in derselben Datei vereinbart wurde (in Zeile 31 wurde sie **deklariert**).

Aufgabe 1: Die Dateien `sVereinbarungen01.c` und `sVereinbarungen02.c` werden von vielen C-Ausführern als Programm akzeptiert, obwohl sie unvollständig sind: In diesen Dateien wird eine Funktion deklariert und benutzt (aufgerufen), aber nicht definiert. Der Ausführer "ergänzt" die fehlende Funktionsdefinition aus seiner Standardbibliothek. Um welche Funktion handelt es sich? Wo wird sie deklariert? Wo wird sie benutzt?

Den **drei** deutschen Worten **Vereinbarung**, **Deklaration** und **Definition** entsprechen im Englischen nur **zwei** Worte: **declaration** und **definition**. Die folgenden Zeilen sollen die korrekte Übersetzung dieser Worte verdeutlichen:

Eine **Vereinbarung** ist	A **declaration** is
entweder eine **Deklaration**	either a **declaration** which reserves no storage
oder eine **Definition**.	or a **definition**.

7 Funktionsdefinitionen und lokale ("interne") Vereinbarungen

Unter allen externen **Vereinbarungen**, die in den Dateien eines C-Programms stehen können, spielen die Definitionen von **Funktionen** eine ganz besonders wichtige Rolle. Denn in den Funktionsdefinitionen stehen (fast) alle **Anweisungen** und **Ausdrücke**, aus denen das Programm besteht.

Def.: Eine **Anweisung** (engl. statement) ist ein Befehl (des Programmierers an den Ausführer), den Wert bestimmter Variablen zu verändern ("Tue die Werte ... in die Variablen ...").

Def.: Ein **Ausdruck** (engl. expression) ist ein Befehl (des Programmierers an den Ausführer), einen Wert zu berechnen.

Def.: Eine **Definition** ist ein Befehl (des Programmierers an den Ausführer), eine Größe zu erzeugen (z.B. eine Variable oder eine Funktion).

Funktionsdefinitionen sind die einzigen Vereinbarungen, die andere Vereinbarungen enthalten können. Diese bezeichnet man als **lokale** (oder interne) **Vereinbarungen**, um sie von den externen Vereinbarungen, die direkt in einer Datei stehen, zu unterscheiden.

Das Beispiel 1 des vorigen Abschnitts enthält (neben zahlreichen externen Vereinbarungen) auch zwei lokale Vereinbarungen.

Innerhalb einer Funktionsdefinition darf man Funktionen deklarieren aber nicht **definieren**. Somit sind alle Funktionsdefinitionen externe Vereinbarungen. Man sagt auch: Funktionsdefinitionen dürfen nicht geschachtelt werden. Im Beispiel 2 des vorigen Abschnitt wird innerhalb der `main`-Funktion eine Funktion namens `printf` deklariert (siehe dort Zeile 22).

8 Der Präprozessor und der Compiler

Die Sprache C wurde vor allem in den 1960-er Jahren entwickelt. Damals waren typische Computer um einen Faktor von etwa hunderttausend oder eine Million langsamer als heute (im Jahr 2003) übliche PCs. Außerdem war der Hauptspeicher eines Computers häufig so klein, dass ein größeres Quellprogramm nicht vollständig hineinpasste, sondern "portionsweise" eingelesen und übersetzt werden musste. Deshalb lag es nicht nahe, beim Prüfen und Übersetzen einer Quelldatei auch noch weitere Dateien zu berücksichtigen. Das grundlegende Modell, nach dem C-Quellprogramme auch heute noch in ein ausführbares Programm übersetzt werden, sind stark von den damaligen Beschränkungen geprägt.

Ein C-Programm besteht aus **Quelldateien** (source files oder preprocessing files). Eine Quelldatei kann zusätzlich zu C-Befehlen auch so genannte **Präprozessor-Befehle** enthalten, die alle mit einem Nummernzeichen # beginnen. Das Compilieren einer Quelldatei erfolgt mit Hilfe von zwei Programmen: einem **Präprozessor** und dem eigentlichen **Compiler** und besteht aus den folgenden beiden Schritten:

1.1. Der **Präprozessor** übersetzt die **Quelldatei** in eine **Übersetzungseinheit** (translation unit).
1.2. Der **Compiler** übersetzt die **Übersetzungseinheit** in eine **Objektdatei**.

Der Präprozessor ist ein ziemlich **mächtiges** und gleichzeitig sehr **primitives** Programm. Er kann nur wenige (Präprozessor-) Befehle ausführen, achtet dabei nicht auf die Syntaxregeln von C und "weiß nicht" was ein Typ ist. Man benützt ihn vor allem, um so genannte **Kopfdateien** (header files) in eine Quelldatei zu kopieren (mit dem `#include`-Befehl, siehe nächsten Abschnitt), um so genannte Makros zu definieren und um Teile einer Quelldatei nur unter bestimmten Bedingungen an den Compiler weiter zureichen (z.B. nur wenn das Symbol TEST definiert ist oder wenn die Übersetzung unter einem bestimmten Betriebssystem erfolgt). Mit dem Präprozessor kann man eine Reihe wichtiger Probleme lösen, er ist aber auch schon mehrfach mit Erfolgt dazu verwendet worden, C-Programme weitgehend **unlesbar** zu gestalten.

9 Kopfdateien (header files)

Besteht ein C-Programm aus **mehreren** Dateien und übergibt man **eine** davon dem Ausführer (indem man die Datei compiliert), dann prüft der Ausführer grundsätzlich nur diese **eine** Datei und berücksichtigt dabei keine anderen Dateien (auch wenn man ihm diese Dateien bereits übergeben hat). Hat man in einer Datei `d1` eine Variable oder Funktion `g` **definiert** und will man `g` in anderen Dateien `d2`, `d3`, ... **benutzen**, so muss man `g` in diesen anderen Dateien **deklarieren**.

Def.: Eine **Deklaration** ist ein **Versprechen** (des Programmierers an den Ausführer), dass eine bestimmte Größe (Variable oder Funktion) in irgendeiner Datei eines Programms **definiert** wird.

Beispiel 1: Größen in einer Datei **definieren** und in anderen Dateien **deklarieren und benutzen**

```
 1 // Datei dat01.c
 2
 3   int    summe    (int otto, int emil); // Deklaration (mit  Param-Namen)
 4   float quotient(int,   int);          // Deklaration (ohne Param-Namen
 5 //float quotient(float, float);        // Deklaration, falsche!
 6   int    printf  (const char *, ...);  // Deklaration, richtige!
 7
 8 unsigned int anzahlAufrufe;            // Definition
 9
10 int main(void) {                       // Definition
11    printf("summe    (+5, -3): %+7d\n",    summe    (+5, -3));
12    printf("quotient(+7, +2): %+7.2f\n", quotient(+7, +2));
13    printf("Anzahl Aufrufe:   % 7d\n",    anzahlAufrufe);
14    return 0;
15 } // main
```

```
16 // Datei dat02.c
17
18 extern unsigned int anzahlAufrufe;
19
20 int summe(int n1, int n2) {
21    anzahlAufrufe++;
22    return n1 + n2;
23 } // summe
24
25 float quotient(int n1, int n2) {
26    anzahlAufrufe++;
27    return (float) n1 / (float) n2;
28 } // quotient
```

Die Variable `anzahlAufrufe` **wird in der Datei** `dat01.c` **definiert** (Zeile 8) und in der Datei `dat02.c` **deklariert** (Zeile 18) und **benutzt** (Zeilen 21 und 26).

Aufgabe 1: Wo wird die Funktion `summe` definiert? Wo wird sie deklariert? Wo wird sie benutzt? Ebenso für die Funktion `quotient`.

In Zeile 4 wird die Funktion `quotient` korrekt deklariert, d.h. als Funktion mit 2 `int`-Parametern. In Zeile 5 wird die Funktion `quotient` falsch deklariert, nämlich als Funktion mit 2 `float`-Parametern (die Zeile 5 ist oben "auskommentiert" und somit unwirksam).

Angenommen, dem Programmierer unterläuft ein Flüchtigkeitsfehler und er deklariert die Funktion `quotient` wie in Zeile 5 statt wie in Zeile 4. Dann akzeptieren viele C-Ausführer das Programm ohne Fehlermeldung oder Warnung, aber Aufrufe der Funktion `quotient` in der Datei `dat01.c` (etwa der Aufruf in Zeile 12) liefern falsche Ergebnisse. Die Ausgabe des obigen Beispielprogramms (Quelldateien `dat01.c` und `dat02.c`) sieht dann etwa wie folgt aus:

```
29 -------------------------
30 summe    (+5, -3):      +2
31 summe    (-2, -4):      -6
32 -------------------------
33 quotient(+7, +2):   +1.01
34 quotient(-7, +4):   -0.98
35 quotient(+7, -4):   -1.02
36 -------------------------
37 Anzahl Aufrufe:         5
```

Nur durch sehr sorgfältiges Testen kann der Programmierer seinen Flüchtigkeitsfehler entdecken.

Um solche Deklarationsfehler unwahrscheinlich zu machen, hat man die so genannten **Kopfdateien** (header files) erfunden. Wenn man in einer **Quelldatei** namens `dat.c` Größen definiert, die auch in anderen Dateien benutzt werden sollen, schreibt man Deklarationen dieser Größen üblicherweise in eine **Kopfdatei** namens `dat.h`, etwa so:

Beispiel 2: Größen in einer Datei **definieren** und in anderen Dateien **benutzen** (mit Kopfdateien)

```
 1 // Datei dat01.h
 2
 3 extern unsigned int anzahlAufrufe;
```

```
 4 // Datei dat02.h
 5
 6   int   summe   (int otto, int emil); // Mit  Param-Namen
 7   float quotient(int,   int);         // Ohne Param-Namen
 8 //float quotient(float, float);       // Falsche Deklaration
```

```
 9 // Datei dat01.c
10 // ------------------------------------------------------
11 #include "dat01.h"
12 #include "dat02.h"
13 #include <stdio.h>
14 // ------------------------------------------------------
15 unsigned int anzahlAufrufe;         // Anfangswert ist 0
16
17 int main(void) {
18    printf("summe   (+5, -3): %+7d\n",   summe   (+5, -3));
19    printf("quotient(+7, +2): %+7.2f\n", quotient(+7, +2));
20    printf("Anzahl Aufrufe:   % 7d\n",   anzahlAufrufe);
21    return 0;
22 } // main
```

```
23 // Datei dat02.c
24 // ------------------------------------------------------
25 #include "dat01.h"
26 #include "dat02.h"
27 // ------------------------------------------------------
28 int summe(int n1, int n2) {
29    anzahlAufrufe++;
30    return n1 + n2;
31 } // summe
32
33 float quotient(int n1, int n2) {
34    anzahlAufrufe++;
35    return (float) n1 / (float) n2;
36 } // quotient
```

Die Quelldatei `dat01.c` enthält drei Präprozessor-Befehle (in den Zeilen 11 bis 13). Der Präprozessor ersetzt jeden `#include`-Befehl durch den Inhalt der darin genannten Datei. Der Unterschied zwischen den Notationen `#include <stdio.h>` und `#include "dat01.h"` ist ziemlich simpel: Nach der Datei `stdio.h` sucht der Präprozessor nur in bestimmten Bibliotheksverzeichnissen. Nach der Datei `dat01.h` sucht er dagegen zuerst im aktuellen Arbeitsverzeichnis, und nur wenn er sie dort nicht findet auch in den Bibliotheksverzeichnissen.

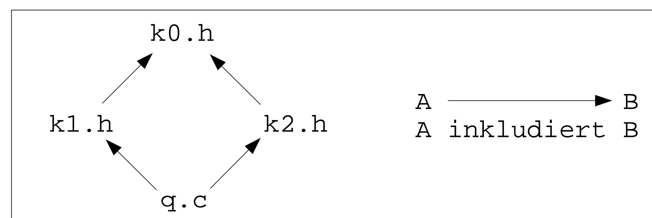
Mit dem `#include`-Befehl kann man **beliebige** Dateien in eine Quelldatei kopieren lassen, es werden **keinerlei Prüfungen** durchgeführt. Ein `#include`-Befehl verändert nicht die original-Quelldatei, in der er steht, sondern nur eine temporäre Kopie davon (die veränderte temporäre Kopie wird anschließend an

den Compiler geschickt). Die kopierte Datei darf weitere `#inlcude`-Befehle enthalten; auch die werden durch die Inhalte der betreffenden Dateien ersetzt. Wenn zwei Dateien sich "gegenseitig inkludieren", geraten manche Präprozessoren in eine Endlosrekursion (nicht empfehlenswert!).

Aufgabe 1: Was passiert vermutlich, wenn man (im Beispiel 1) in Zeile 11 den Präprozessor-Befehl `#include "dat01.h"` durch den Befehl `#include <dat01.h>` ersetzt? Was passiert wohl, wenn man in Zeile 13 den Befehl `#include <stdio.h>` durch den Befehl `#include "stdio.h"` ersetzt?

Aufgabe 2: Jede Definition gilt auch als Deklaration. Warum inkludiert man trotzdem die Kopfdatei `dat02.h` auch in die Datei `dat02.c`, in der `summe` und `quotient` definiert werden? Welche Fehler würden ohne diesen kleinen Trick vom Ausführer (genauer: vom Compiler) nicht erkannt? Was passiert, wenn man die richtige Deklaration in Zeile 7 auskommentiert und die falsche Deklaration in Zeile 8 gültig macht?

Da jede Datei andere Dateien inkludieren darf, kann es leicht passieren, dass eine Quelldatei `q.c` eine bestimmte Kopfdatei `k0.h` mehr als einmal inkludiert, z.B. in folgendem Fall:



Wenn die Datei `k0.h` nur **Deklarationen** enthält, kann sie ohne Probleme mehrfach inkludiert werden (denn man darf ja jede Größe in jeder Datei beliebig oft deklarieren). Falls die Datei `k01.h` auch **Definitionen** enthält (z.B. Definitionen häufig benötigter Konstanten), muss man verhindern, dass sie mehrfach inkludiert wird (denn man man darf jede Größe in einem Programm nur einmal definieren), etwa so, wie im folgenden Beispiel:

Beispiel 3: Mehrfaches Inkludieren einer Datei in eine andere Datei verhindern

```
 1 // Datei k0.h
 2
 3 #ifndef k0_h
 4 #define k0_h
 5
 6 const int konst0 = 123;
 7
 8 #endif // k0_h
```

```
 9 // Datei k1.h
10
11 #include "k0.h"
```

```
12 // Datei k2.h
13
14 #include "k0.h"
```

```
15 // Datei q1.c
16 // ------------------------------------------------------------
17 #include "k1.h"
18 #include "k2.h"
```

```
19 #include <stdio.h>
20 // ------------------------------------------------------------
21 int main(void) {
22    printf("------------------------------------------------\n");
23    printf("prog (q1.c, q2.c, k0.h, k1.h, k2.h): Los geht's!\n");
24    printf("konst0: %d\n", konst0);
25    printf("------------------------------------------------\n");
26    return 0;
27 } // main
28 // ------------------------------------------------------------
```

```
29 // Datei q2.c
30 // ------------
31 #include "k1.h"
32 ...
```

Jedes Mal, wenn der Präprozessor den Befehl `#include k0.h` ausführt, prüft er, ob das Symbol `k0_h` definiert ist oder nicht (siehe Zeile 3). Nur wenn es noch nicht **definiert** ist, definiert er es (Zeile 4) und kopiert alle Zeilen bis zum nächsten `#endif` (in Zeile 8). Dadurch wird verhindert, dass durch die beiden `#include`-Befehle in Zeile 17 und 18 die Datei `k0.h` mehr als einmal in die Datei `q1.c` kopiert wird (denn nach dem ersten Kopiervorgang ist das Symbol `k0_h` definiert).

Mit diesem Trick kann man allerdings nur verhindern, dass die Kopfdatei `k0.h` mehrmals in dieselbe (Quell-) Datei kopiert wird. Man kann damit nicht verhindern, dass die Kopfdatei `k0.h` einmal in die Quelldatei `q1.c` und dann noch mal in die Quelldatei `q2.c` inkludiert wird. Dadurch wird die Konstante `konst0` innerhalb des Gesamtprogramms **zweimal definiert** und der Binder gibt eine entsprechende Fehlermeldung oder eine Warnung aus, etwa so:

Ausgabe des Borland-Compilers:

```
bcc32 -eprog.exe q1.c q2.c
Borland C++ 5.5.1 for Win32 Copyright (c) 1993, 2000 Borland
q1.c:
q2.c:
Turbo Incremental Link 5.00 Copyright (c) 1997, 2000 Borland
Warning: Public symbol '_konst0' defined in both module
D:\MEINEDATEIEN\BSPC\INCLUDEMEHRFACH\Q1.OBJ and
D:\MEINEDATEIEN\BSPC\INCLUDEMEHRFACH\Q2.OBJ
prog
------------------------------------------------
prog (q1.c, q2.c, k0.h, k1.h, k2.h): Los geht's!
konst0: 123
------------------------------------------------
```

Ausgabe des C-Compilers lcc-win32:

```
lcc    -c q1.c
lcc    -c q2.c
lcclnk -o prog q1.obj q2.obj
q2.obj: multiple definition of _konst0
first definition is in file q1.obj
prog
------------------------------------------------
prog (q1.c, q2.c, k0.h, k1.h, k2.h): Los geht's!
konst0: 123
------------------------------------------------
```

Ausgabe des Gnu-C-Compilers gcc:

```
gcc -c q1.c
gcc -c q2.c
gcc -o prog.exe q1.o q2.o
```

```
q2.o(.text+0x0):q2.c: multiple definition of `_konst0'
q1.o(.text+0x0):q1.c: first defined here
collect2: ld returned 1 exit status
make: *** [prog.exe] Error 1
```

Die Compiler `bcc32` und `lcc` erzeugen trotz des Fehlers eine ausführbare Datei, der `gcc` meldet nur einen Fehler und weigert sich, die falschen Objektdateien zu binden.

Zusammenfassung: Kopfdateien dienen dazu, bestimmte Schwächen des C-Ausführers zu kompensieren. Die wichtigste Schwäche besteht darin, dass der Compiler keine Typprüfungen über Dateigrenzen hinaus durchführt. Deshalb muss der Programmierer alle für die Prüfung einer Quelldatei benötigten Typinformationen dem Compiler in dieser Datei zur Verfügung stellen, indem er selbst von Hand entsprechende Deklarationen in die Quelldatei schreibt (was mühsam und fehleranfällig ist) oder durch `#include`-Befehle aus Kopfdateien kopieren lässt (was weniger mühsam und weniger fehleranfällig ist).

Eine Quelldatei q zusammen mit allen Dateien, die von q inkludiert werden, bezeichnet man auch als eine **Präprozessor-Übersetzungseinheit** (preprocessing translation unit). Der Präprozessor übersetzt also **Präprozessor-Übersetzungseinheiten in Übersetzungseinheiten**.

Anmerkung: Unter anderem weil C-Compiler immer nur "eine Datei auf einmal" prüfen und übersetzen und keine Typprüfungen über Dateigrenzen hinaus durchführen, können sie deutlich kleiner und einfacher sein, als andere Compiler. Trotzdem: Auch in der Informatik geht der Trend dahin, möglichst viel menschliche Arbeit (z.B. Arbeit des Programmierers) durch maschinelle Arbeit (z.B. Arbeit eines Compilers) zu ersetzen.

Anmerkung: Das Konzept einer Kopfdatei gibt es nur in C (und in C++), aber in keiner anderen verbreiteten Programmiersprache.

10 Der Binder

Der `#include`-Befehl ist nur ein simpler Kopierbefehl, mit dem man eine beliebige Datei (z.B. eine Kopfdatei oder irgendeine andere Datei) in eine beliebige Datei (z.B. in eine C-Quelldatei oder in irgendeine andere Datei) kopieren kann. Der `#include`-Befehl "weiß nicht, wie ein C-Programm aussehen sollte" und führt keinerlei Prüfungen durch. Typischerweise kopiert man mit dem `#include`-Befehl Kopfdateien (die typischerweise nur Deklarationen enthalten) in eine C-Quelldatei, in der man die deklarierten Größen verwenden will.

Dem Binder (link program, "Linker") kann man befehlen, eine oder mehrere Objektdateien zu einem ausführbaren Programm zusammen zubinden. Der Binder prüft dann, ob alle in den Objektdateien verwendeten Größen (Variablen und Funktionen) in **genau einer** Objektdatei definiert wurden. Falls eine benutzte Größe mehr als einmal definiert wurde, meldet der Binder einen Fehler.

Falls eine verwendete Größe g in keiner der angegebenen Objektdateien definiert ist, sucht der Binder in bestimmten Bibliotheken nach einer Definition. Falls er eine findet, bindet er sie automatisch in das Programm ein. Das gilt typischerweise z.B. für die Definitionen von Standardfunktionen wie `printf`, `exit` und `sin` etc.. Dabei macht es keinen Unterschied, ob der Programmierer diese Größen in seinen Quelldateien "von Hand" deklariert hat oder die Deklarationen mit `#include`-Befehlen aus irgendwelchen Kopfdateien hat kopieren lassen. Entscheidend ist, dass die Größe in mindestens einer Objektdatei deklariert und **benutzt** wird. Falls der Binder keine Definition findet (weder in den angegebenen Objektdateien noch in seinen Bibliotheken) meldet er einen Fehler.

Einfache Binder können nur feststellen, dass eine bestimmte Größe in einer Objektdatei benutzt wird. Sie können aber nicht feststellen, welcher **Zeile** in welcher Quelldatei diese Benutzung entspricht (Objektdateien enthalten häufig keine "Zeileninformationen"). Deshalb enthalten Fehlermeldungen des Binders häufig keine Zeilenangaben und sind nicht immer leicht zu verstehen.

Beispiel 1: Fehlermeldungen von Compilern und Bindern:

In der einzigen Quelldatei `Hexer01.c` eines Programms wurde der folgende Aufruf einer nicht-definierten und nicht-deklarierten Funktion namens `sub` eingefügt:

```
122 ...
123 sub(1, 2);
124 ...
```

Der **Gnu-C-Compiler** meldet (wenn man ihn ohne spezielle Optionen auf die Datei `Hexer01.c` anwendet) keinen Fehler, aber der Binder gibt uns folgende Meldung aus:

```
1 /cygdrive/c/dokus/Temp/ccgibASt.o(.text+0x602):Hexer01.c:
2 undefined reference to `_sub'
3 collect2: ld returned 1 exit status
```

Dabei ist `ccgibAst.o` ein Dateiname, den der C-Ausführer erfunden hat und das Verzeichnis `/cygdrive/c/dokus/Temp/` ist ein internes Verzeichnis des Ausführers (kein Verzeichnis des Programmierers).

Der **Borland-C/C++-Compiler** `bcc32` und der **C-Compiler** `lcc-win32` melden beide einen Fehler in Zeile 123 der Quelldatei `Hexer01.c`, weil die Funktion `sub` nicht deklariert wurde. Fügt man eine geeignete Deklaration ein, melden die Compiler keinen Fehler mehr. Der Binder `lcclnk` (der zum Compiler `lcc-win32` gehört) gibt dann die folgende, sehr schlichte Fehlermeldung aus:

```
33 hexer01.obj .text: undefined reference to '_sub'
```

Die Fehlermeldung des **Borland-Binders** ist etwas länger, aber nicht viel informativer:

```
34 Turbo Incremental Link 5.00 Copyright (c) 1997, 2000 Borland
35 Error: Unresolved external '_sub' referenced from
36 D:\MEINEDATEIEN\BSPC\HEXER\HEXER01.OBJ
```

Wenn ein Programm aus vielen Quelldateien (und nach dem Compilieren aus vielen Objektdateien) besteht, kann es schwierig sein, solche Fehlermeldungen zu verstehen und zu beseitigen.

11 Sichtbarkeit

Bezeichner (identifier) in einem C-Programm dienen dazu, Größen zu bezeichnen, z.B. Variablen oder Funktionen. Ein Bezeichner (z.B. `otto`) kann in verschiedenen Teilen eines Programms unterschiedliche Größen bezeichnen (z.B. eine `float`-Variable und eine `int`-Variable).

Eine Vereinbarung (Deklaration oder Definition) ist im allgemeinen nicht im ganzen Programm sichtbar (visible), sondern nur in einem bestimmten Bereich, dem Gültigkeitsbereich (scope) der Vereinbarung. Innerhalb ihres Gültigkeitsbereichs kann man mit dem vereinbarten Bezeichner (z.B. `otto`) die vereinbarte Größe (z.B. eine `float`-Variable) bezeichnen.

Beispiel 1: Dateien als Gültigkeitsbereich

```
1 // Datei dat.c
2 ...
3 float otto = 123.45;
4 ...
```

In diesem Beispiel steht die Vereinbarung direkt in einer Datei (und nicht innerhalb einer Funktionsdefinition). Ihr Gültigkeitsbereich beginnt unmittelbar nach ihrem Text (d.h. nach dem Semikolon in Zeile 3) und erstreckt sich bis zum Ende der Datei `dat.c`. In Zeile 2 ist die Vereinbarung noch nicht sichtbar.

Außerdem ist diese Definition in jeder anderen Datei sichtbar, in der die `float`-Variable `otto` deklariert wird, und zwar unmittelbar nach der ersten Deklaration und bis zum Ende der anderen Datei.

Beispiel 2: Eine Datei als Gültigkeitsbereich

```
 5 // Datei dat.c
 6 ...
 7 static float emil = 123.45;
 8 ...
```

Diese Vereinbarung ist nur innerhalb der Datei `dat.c` (und in keiner anderen Datei) sichtbar. Ihr Gültigkeitsbereich beginnt unmittelbar nach ihrem Text, in Zeile 6 ist sie somit noch nicht sichtbar.

Beispiel 3: Ein Block als Gültigkeitsbereich

```
 9 int summe(float f1, float f2) {
10    ...
11    float otto = f1 + f2;
12    ...
13 }
14 ...
```

In diesem Beispiel steht die Vereinbarung der Variablen `otto` in einem Block (genauer: Im Rumpf einer Funktionsdefinition). Ihr Gültigkeitsbereich beginnt unmittelbar nach ihrem Text (d.h. nach dem Semikolon in Zeile 11) und erstreckt sich bis zum Ende des Blocks. In Zeile 10 ist die Vereinbarung noch nicht und in Zeile 14 nicht mehr sichtbar.

Beispiel 4: Geschachtelte Blöcke als Gültigkeitsbereich

```
15 int summe(float f1, float f2) {
16    ...
17    {
18       ...
19       float otto = f1 + f2;
20       ...
21       {
22          ...
23          int otto = 17;
24          ...
25       }
26       ...
27    }
28 } // summe
```

Die Vereinbarung der `float`-Variablen `otto` in Zeile 19 ist sichtbar in Zeile 20 bis 22 und 26. Sie ist noch nicht sichtbar in Zeile 16 bis 18 . Außerdem ist sie in Zeile 23 und 24 nicht sichtbar.

Die Vereinbarung der `int`-Variablen `otto` in Zeile 23 ist sichtbar in Zeile 24.

Aufgabe 1: Was gibt das Programm aus, welches aus den folgenden beiden Dateien `sicht01.c` und `sicht02.c` besteht? In welchen Zeilen ist die Vereinbarung der `int`-Variablen `n01` (in Zeile 54) sichtbar?

```
 1 // Datei sicht01.C
 2
 3 #include <stdio.h>
 4 // ------------------------------------------------------
 5 extern int n01;
 6 extern int n02;
 7 extern int n03;
 8 extern int n04;
 9
10 extern char * f01(void);
11 extern char * f02(void);
12 extern char * f03(void);
13 extern char * f04(void);
14 // ------------------------------------------------------
15 void gibWasAus() {
16    printf("A n01:   %d\n", n01);
17    int n01 = 112;
18    printf("B n01:   %d\n", n01);
19    {
20       printf("C n01:   %d\n", n01);
21       int n01 = 113;
22       printf("D n01:   %d\n", n01);
23    }
24    printf("E n01:   %d\n", n01);
25    {
26       printf("F n01:   %d\n", n01);
27       int n01 = 114;
28       printf("G n01:   %d\n", n01);
29    }
30    printf("H n01:   %d\n", n01);
31 } // gibWasAus
32 // ------------------------------------------------------
33 int main(void) {
34    printf("------------------------\n");
35    gibWasAus();
36    printf("------------------------\n");
37
38    n03++; // n03 ist eigentlich const, aber ...
39
40    printf("I n01:   %d\n", n01);
41 // printf("J n02:   %d\n", n02);
42    printf("K n03:   %d\n", n03);
43 // printf("L n04:   %d\n", n04);
44    printf("------------------------\n");
45    printf("M f01(): %s\n", f01());
46 // printf("N f02(): %s\n", f02());
47    printf("O f03(): %s\n", f03());
48 // printf("P f04(): %s\n", f04());
49    printf("------------------------\n");
50    return 0;
51 } // main
```

```
52 // Datei sicht02.c
53 // ------------------------------------------------------
54                  int n01 = 111;
55    static        int n02 = 222;
56           const int n03 = 333;
57    static const int n04 = 444;
58
59                  char * f01(void) {return "f01";}
60    static        char * f02(void) {return "f02";}
```

```
61           const char * f03(void) {return "f03";}
62    static const char * f04(void) {return "f04";}
```

Wenn man versucht, aus diesen beiden Dateien ein ausführbares Programm zu erzeugen und dieses Programm ausführen zu lassen (falls seine Erzeugung gelingt), erhält man folgende Ausgaben:

Vom Borland-Compiler bcc32:

```
bcc32 -esicht.exe sicht01.c sicht02.c
Borland C++ 5.5.1 for Win32 Copyright (c) 1993, 2000 Borland
sicht01.c:
Error E2140 sicht01.c 17: Declaration is not allowed here in function gibWasAus
Error E2140 sicht01.c 21: Declaration is not allowed here in function gibWasAus
Error E2140 sicht01.c 27: Declaration is not allowed here in function gibWasAus
*** 3 errors in Compile ***
```

Vom C-Compiler lcc-win32:

```
lcc    -c sicht01.c
lcc    -c sicht02.c
lcclnk -o sicht.exe sicht01.obj sicht02.obj
sicht.exe
-------------------------
A n01:   111
B n01:   112
C n01:   112
D n01:   113
E n01:   112
F n01:   112
G n01:   114
H n01:   112
-------------------------
I n01:   111
K n03:   334
-------------------------
M f01(): f01
O f03(): f03
-------------------------
```

Vom Gnu-C-Compiler gcc:

```
gcc -c sicht01.c
gcc -o sicht.exe sicht01.o sicht02.o
sicht.exe
make: *** [sicht.exe] Segmentation fault (core dumped)
make: *** Deleting file `sicht.exe'
```

12 Vorvereinbarte und vom Programmierer vereinbarte Typen

Zur Sprache C gehören eine Reihe festgelegter **Typnamen** wie `int`, `unsigned int`, `long`, `float` etc. und zu jeder Implementierung von C gehören mehr oder weniger viele vordeklarierte **Typen** (mindestens ein Ganzzahltyp und mindestens ein Gleitpunkttyp).

Außerdem gibt es zu jedem Typ T zahlreiche **abgeleitete Typen** (derived types), zum Typ `int` z.B. die folgenden: `int[]` ("Reihung von int"), `int[][]` ("Zweidimensionale Reihung von `int`"), `int *` ("Adresse von `int`") etc.

Der Programmierer kann zusätzliche Bezeichner für schon vorhandene Typen und neue Typen vereinbaren.

Beispiel 1: Zusätzliche Namen für schon vorhandene Typen vereinbaren

```
 1 typedef int ganz
 2 typedef unsigned int u_ganz
 3 typedef char * string
 4
 5 int          n01 = 17;
 6 ganz         n02 = n01;
 7
 8 unsigned int n03 = 25;
 9 u_ganz       n04 = n03;
10
11 char *       s01 = "Hallo Sonja!";
12 string       s02 = s01;
```

Nach diesen Vereinbarungen hat der Typ `int` zusätzlich den Namen `ganz` und dem Ausführer ist es gleichgültig, welchen der beiden Namen man verwendet (d.h. die beiden Variablen n01 und n02 gehören **zum selben Typ**). Der Typ `unsigned int` hat zusätzlich den Namen `u_ganz` und anstelle des Typbezeichners `char *` ("Adresse von `char`") kann man auch den Bezeichner `string` verwenden.

Anmerkung 1: Durch die Vereinbarungen in den Zeilen 1 bis 3 werden **keine neuen Typen** vereinbart, nur zusätzliche **Namen** für schon vorhandene Typen.

Anmerkung 2: Im Zusammenhang mit **Typbezeichnern** und **Typen** wird in C (anders als in C++) im allgemeinen **nicht** genau zwischen **Deklarationen** und **Definitionen** unterschieden. Im Deutschen bietet es sich an, einfach von **Vereinbarungen** zu sprechen.

Ein C-Programmierer kann nicht nur neue Typbezeichner, sondern auch **neue Typen** vereinbaren, und zwar so genannte **Aufzählungstypen** (enumeration types), **Verbundtypen** (struct types) und **Vereinigungstypen** (union types). Ein C-Programmierer kann **keine** neuen Ganzzahltypen (wie `short` oder `int`) und keine neuen Gleitpunkttypen (wie `float` oder `double`) vereinbaren (wie das z.B. in der Sprache Ada möglich ist).

Es folgen schon hier die wichtigsten Regeln zum Vereinbaren von Typen:

TypVer01: Jeden (nicht vorvereinbarten) **Typbezeichner** und jeden (nicht vorvereinbarten) **Typ** muss man in **jeder Datei**, in der man ihn benutzen will, vereinbaren.

TypVer02: Innerhalb einer Datei darf man jeden Typbezeichner oder Typ höchstens **einmal** vereinbaren (nicht mehrmals).

TypVer03: Wenn der Programmierer in verschiedenen Dateien **unterschiedliche** Typen mit **gleichen** Namen vereinbart, dann ist das im allgemeinen ein Fehler, den häufig weder der Compiler noch der Binder entdeckt (siehe dazu das Beispielprogramm `GleicheNamen`).

Wegen **TypVer03** sollte man Typen stets in Kopfdateien vereinbaren und mit `#include`-Befehlen in die Dateien kopieren lassen, in denen man die Typen benutzen will. Wegen **TypVer02** muss man verhindern, dass eine solche Kopfdateie mehrfach in eine Quelldatei kopiert wird.

13 Aufzählungstypen (enum types)

Beispiel 1: Vereinbarung und Benutzung von Aufzählungstypen

```
 1 #define FARBE enum Farbe
 2 // ------------------------------------------------------------------------
 3                                  // Name(n) des neuen Aufzaehlungs-Typs:
 4 enum Bool    {FALSE, TRUE};      // "enum Bool"
 5 enum Farbe   {ROT, GRUEN, BLAU}; // "enum Farbe"  oder  "FARBE"
 6 // ------------------------------------------------------------------------
 7 enum Bool  b01 = FALSE;
 8 enum Bool  b02 = TRUE;
 9
10 enum Farbe f01 = ROT;
```

```
11 enum Farbe f02 = BLAU;
12 FARBE      f03 = GRUEN;
13 FARBE      f04 = ROT;
14 // ------------------------------------------------------------------------
15 FARBE FarbeSucc(FARBE f) {
16    // Verlaesst sich darauf, dass f einen Farbe-Code enthaelt.
17    // Liefert den Code der naechsten Farbe (zyklisch, nach BLAU kommt ROT)
18    if (f==BLAU) return ROT;
19    return f+1;
20 } // FarbeSucc
```

Durch die Typvereinbarung in Zeile 4 wird bei den meisten C-Ausführern folgendes bewirkt:

Der Name `enum Bool` ist ein zusätzlicher Name für den Typ `int`, FALSE (bzw. TRUE) bezeichnet eine `int`-Konstante mit dem Wert 0 (bzw. 1). Die Variablen `b01` und `b02` sind ganz normale `int`-Variablen, denen man beliebige `int`-Werte (nicht nur 0 und 1) zuweisen kann.

Ganz entsprechend sind `enum Farbe` und (durch den #define-Befehl in Zeile 1) `FARBE` zusätzliche Namen für den Typ `int`. `ROT`, `GRUEN` und `BLAU` sind `int`-Konstanten mit den Werten 0, 1 bzw. 2.

Ein C-Ausführer dürfte prüfen (und verbieten), dass man einer `FARBE`-Variablen andere Werte als `ROT`, `GRUEN` oder `BLAU` zuweist, aber die meisten C-Ausführer scheuen sich, den Programmierer derart einzuschränken.

Diese und weitere Beispiel zum Thema Aufzählungstypen findet man in der Datei `Enum01.c`.

14 Verbundtypen (struct types)

Ein Verbund ist eine Zusammenfassung von (null oder mehr) Variablen, die nicht notwendig zum selben Typ gehören. Verbunde sind die Vorgänger von Objekten in Sprachen wie C++ und Java.

Beispiel 1: Vereinbarung und Benutzung von Verbundtypen

```
 1 // ------------------------------------------------------------------------
 2 struct Person {                      // Der Typ heisst "struct Person"
 3    unsigned short jahre;
 4    unsigned short zentimeter;
 5    float          kilo;
 6 };
 7
 8 struct Person p01 = {23, 175, 72.5};
 9 struct Person p02 = {12, 142, 51.0};
10 // ------------------------------------------------------------------------
11 struct Bruch {                       // Der Typ heisst "struct Bruch"
12    int      zaehler;
13    unsigned nenner;
14 } b01;                               // "b01" bezeichnet eine Variable!
15
16 struct Bruch b02 = {-123, 17};
17 struct Bruch b03 = {3333,  5};
18 // ------------------------------------------------------------------------
19 typedef struct {
20    char * vor;
21    char * nach;
22 } NAME;                              // "NAME" bezeichnet einen Typ!
23
24 NAME n01 = {"Karl", "Schmitz"};
25 NAME n02 = {"Ali",  "Oezdemir"};
26 // ------------------------------------------------------------------------
27 typedef struct Komplex {             // Der Typ heisst "struct Komplex" und
```

```
28    double re;
29    double im;
30 } KOMPLEX;                          // zusaetzlich auch "KOMPLEX"
31
32 struct Komplex k01 = {1.0, 2.0};
33 struct Komplex k02 = {2.5, 1.5};
34
35 KOMPLEX        k03 = {3.333, 6.666};
36 KOMPLEX        k04;
37 // ------------------------------------------------------------------------
38 void gibAusPerson(struct Person p) {
39    printf("D Jahre: %3d, Zentimeter: %3d, Gewicht: %4.1f\n",
40              p.jahre,    p.zentimeter,    p.kilo);
41 } // gibAusPerson
42 // ------------------------------------------------------------------------
43 struct Person jahrePlus1(struct Person p) {
44    p.jahre++;
45    return p;
46 } // jahrePlus1
47 // ------------------------------------------------------------------------
```

Diese und weitere Beispiele zum Thema Verbundtypen findet man in der Datei `Verbunde01.c`.

Beispiel 2: Speicher reservieren und wieder freigeben für große Verbunde

```
 1 // Datei Verbunde02.c
 2 #include <stdio.h>  // fuer printf
 3 #include <stdlib.h> // fuer malloc, calloc, free
 4 #include <string.h> // memcopy, memmove, strcpy, strcat, ...
 5
 6 #define LEN 1000 * 1000 * 40
 7 // ------------------------------------------------------------------------
 8 // Jede Variable des Typs VIEL_TEXT belegt ziemlich viel Speicher:
 9 typedef struct {
10    char text1[LEN];
11    char text2[LEN];
12    char text3[LEN];
13 } VIEL_TEXT;
14 // ------------------------------------------------------------------------
15 // Eine Art Konstruktor VIEL_TEXT_K fuer den Verbundtyp VIEL_TEXT (er
16 // liefert einen Zeiger auf einen neuen VIEL_TEXT-Verbund):
17 VIEL_TEXT * VIEL_TEXT_K(char text1[], char text2[], char text3[]) {
18    int    i;
19    char * ind;
20
21    // Speicherplatz fuer eine neue VIEL_TEXT-Variable reservieren:
22    VIEL_TEXT * erg = (VIEL_TEXT *) malloc(sizeof(VIEL_TEXT));
23
24    // In jedes 10. Byte der neuen Variablen das Zeichen '\0' schreiben:
25    ind = (char *) erg;
26    for (i=0; i<sizeof(VIEL_TEXT); i+=10) ind[i]='\0';
27
28    // Ein paar (vermutlich kurze) Texte in die Komponenten der neuen
29    // Verbundvariable schreiben:
30    strcpy(erg->text1, text1);
31    strcpy(erg->text2, text2);
32    strcpy(erg->text3, text3);
33
34    return erg;
35 } // VIEL_TEXT
36 // ------------------------------------------------------------------------
37 int main() {
38    int         i1;
```

```
39     VIEL_TEXT * vp1;
40     printf("i1: ");
41     for (i1=0; i1<ANZ; i1++) {
42        vp1 = VIEL_TEXT_K("Hallo!", "Wie geht's?", "Danke, gut!");
43        printf("%d ", i1);
44        fflush(stdout);
45        free(vp1);                         // Speicher freigeben!
46     }
47
48     return 0;
49  } // main
```

15 Adressen, der Adressoperator & und der Zieloperator *

Def.: Eine **Variable** besteht mindestens aus einer **Adresse** und einem **Wert**. Zusätzlich kann eine Variable einen **Namen** und/oder einen **Zielwert** haben.

Anmerkung: In Java besteht eine Variable (mind.) aus einer **Referenz** und einem **Wert**. Es ist sinnvoll, abstrakte Java-Referenzen und konkretere C-Adressen begrifflich zu unterscheiden.

Sei T irgendein Typ. Dann bezeichnet `T *` (lies: Adresse von T) einen **Adresstyp**. Der Wert einer Variablen eines Adresstyps ist entweder gleich `NULL` oder die Adresse einer Variable des Typs T.

Beispiel 1: Eine "normale" Variable und eine Adressvariable

```
 1 int   otto = 17;   // "Normale" Variable, Typ int
 2 int * emil = NULL; // Adressvariable,     Typ Adresse-von-int
```

Man beachte, dass der Stern `*` in Zeile 2 ein Bestandteil des **Typnamens** ist, und keinen Operator bezeichnet. Ein Adresstyp ist ein Typ, dessen Name mit einem Stern endet.

Weitere Beispiele für Adresstypen sind `float *` (Adresse von `float`), `int * *` (Adresse von Adresse von `int`), `float * * *` (Adresse von Adresse von Adresse von `float`) etc.

Sei `v` irgendeine Variable. Dann bezeichnet `&v` die **Adresse** dieser Variablen und die Variable `v` ist das **Ziel** der Adresse `&v`. Das Ampersand `&` wird in diesem Zusammenhang als **Adressoperator** (engl. address operator) bezeichnet.

Beispiel 2: Eine Adressvariable mit einem Ziel

```
 3 int   zafer = 25;
 4 int * adi   = &zafer;
```

Die Variable `adi` wird hier mit der **Adresse** der Variablen `zafer` initialisiert. Dadurch wird die Variable `zafer` zum **Ziel** der Adressvariablen `adi`. Wenn eine Adressvariable den Wert `NULL` hat (siehe `emil` im vorigen Beispiel) hat sie **kein** Ziel.

Eine Variable vom Typ Adresse-von-int sollte nur Adressen von `int`-Variablen (oder den Wert NULL) enthalten. Eine Variable vom Typ Adresse-von-float sollte nur Adressen von `float`-Variablen (oder den Wert NULL) enthalten etc. In vielen Fällen sorgt der C-Ausführer dafür, dass diese Typregeln eingehalten werden, aber der Programmierer kann ihm befehlen, gegen die Regel zu verstoßen.

Beispiel 3: Erlaubte und nicht erlaubte Werte für Adressvariablen

```
 5 int      i1  = 111;
 6 float    f1  = 1.1;
 7 int    * ai1 = &n1;
 8 int    * ai2 = &f1;            // Typfehler, nicht erlaubt
 9 int    * ai3 = (int *) &f1;    // Typfehler,       erlaubt!
10 float  * af1 = &f1;
11 float  * af2 = &n1;            // Typfehler, nicht erlaubt
12 float  * af3 = (float *) &n1;  // Typfehler,       erlaubt!
```

Variablen eines Adresstyps (wie etwa ai1 und af1) bezeichnen wir hier auch als **Adressvariablen**. Sei av irgendeine Adressvariable, die nicht den Wert NULL hat. Dann bezeichnet * av das Ziel von av, d.h. die Variable, deren Adresse in av steht. Den Stern * bezeichnen wir hier als **Zieloperator**.

Beispiel 4: Adressvariablen und ihre Ziele (Zielvariablen), Fortsetzung von Beispiel 3

```
13                  // Gleichbedeutend mit:
14 int i2;
15 i2   = *ai1;      // i2 = i1;
16 *ai1 = i2 + 3;    // i1 = i2 + 3;
17 *(ai1++);         // i1++;
18
19 float f2;
20 f2   = *af1;      // f2 = f1;
21 *af1 = f2 + 3.0; // f1 = f2 + 3.0;
```

Der **Adressoperator** & liefert die **Adresse** einer Variablen. Der **Zieloperator** * liefert das **Ziel** einer Adresse. Weil man diese Formulierung (hoffentlich) leicht verstehen und sich merken kann, werden hier die Bezeichnungen Adresse, Adressoperator und Zieloperator anderen Bezeichnungen (von denen es viele gibt) vorgezogen.

Achtung: Steht ein Stern * nach einem **Typnamen**, dann ist er selbst Teil eines Typnamens (und **kein** Operator, siehe z.B. Zeilen 7 bis 12). Steht ein Stern vor einer **Variablen**, bezeichnet er den **Zieloperator** (siehe z.B. Zeilen 15 bis 17, 20 und 21). Steht ein Stern zwischen einem Typnamen und einer Variablen, gehört er zum Typnamen (wie z.B. in Zeile 7 bis 12).

Anmerkung: Adressen werden in manchen Büchern auch als **Zeiger**, **Pointer** oder **Referenzen** (und im Englischen als **pointer**, **references** oder **locations**) bezeichnet. Die Bezeichnung **Adressoperator** (für den Operator &) ist üblich und verbreitet. Alternative Bezeichnungen sind **Referenzierungsoperator** und **Zeigeroperator**. Die Bezeichnung **Zieloperator** (für den Operator *) ist nicht verbreitet. Üblich sind die Bezeichnungen **Dereferenzierungsoperator** und **Inhaltsoperator** (im Englischen **dereferencing operator** und **indirection operator**). Jede dieser Bezeichnungen hat Vor- und Nachteile. Formulierungen wie "Der Adressoperator liefert einen Zeiger" (sollte er nicht eine Adresse liefern oder Zeigeroperator heißen?) oder "Den Dereferenzierungsoperator darf man nur auf Pointer anwenden" (sollte er nicht auf Referenzen angewendet werden oder "Entpointerungsoperator" oder so ähnlich heißen?) sind nicht unmittelbar einleuchtend. Der so genannte Inhaltsoperator * liefert nicht nur "den Inhalt einer Adresse", sondern eine Variable, die aus einem Inhalt (Wert) und seiner Adresse besteht. Nur deshalb sind Zuweisungen wie etwa *ai1 = i2 + 3; (siehe Beispiel 3) möglich.

Anmerkung: Es gibt in C auch Adressvariablen, deren Ziel eine **Funktion** ist. Der Zieloperator *, angewendet auf eine solche Adressvariable, liefert natürlich ihr Ziel, d.h. die Funktion.

Beispiel 5: Adressvariablen mit und ohne Typfehler

```
22 // Datei Zeiger01.c
23 #include <stdio.h>   // E/A in Bytestroeme, printf, sprintf, ...
24 // ------------------------------------------------------------------------
25 int     int01 = 1234567890;       // Eine ganz normale   int-Variable
26 float   flt01 = 1.23456789;       // Eine ganz normale float-Variable
27
28 int   * zaii  =           &int01; // Ein Zeiger auf int,   zeigt auf int01
29 int   * zaif  = (int   *) &flt01; // Ein Zeiger auf int  , zeigt auf flt01
30 float * zafi  = (float *) &int01; // Ein Zeiger auf float, zeigt auf int01
31 float * zaff  =           &flt01; // Ein Zeiger auf float, zeigt auf flt01
32 // ------------------------------------------------------------------------
33 int main() {
34    printf("Zeiger01: Jetzt geht es los!\n");
35    printf("-----------------------------------\n");
```

```
36     printf("A                 int01 : %10d\n",
37                               int01);
38     printf("------------------------------------\n");
39     printf("B zaii: %8p, * zaii: %10d\n",
40               zaii,      * zaii);
41     printf("C zafi: %8p, * zafi: %10.2f\n",
42               zafi,      * zafi);
43     printf("------------------------------------\n");
44     printf("D zaif: %8p, * zaif: %10d\n",
45               zaif,      * zaif);
46     printf("E zaff: %8p, * zaff: %10.8f\n",
47               zaff,      * zaff);
48     printf("------------------------------------\n");
49     printf("F                 flt01 : %10.8f\n",
50                               flt01);
51     printf("------------------------------------\n");
52     return 0;
53  } // main
54  /* ------------------------------------------------------------------------
55  Ausgabe des Programms Zeiger01:
56
57  Zeiger01: Jetzt geht es los!
58  ------------------------------------
59  A                  int01 : 1234567890
60  ------------------------------------
61  B zaii: 0x403010, * zaii: 1234567890
62  C zafi: 0x403010, * zafi: 1228890.25
63  ------------------------------------
64  D zaif: 0x403014, * zaif: 1067320914
65  E zaff: 0x403014, * zaff: 1.23456788
66  ------------------------------------
67  F                  flt01 : 1.23456788
68  ------------------------------------
69  ---------------------------------------------------------------------- */
```

Beispiel 6: Eine sinnvolle Anwendung von Adresstypen

```
70  void berechne(int a, int b, int *s, int *d, int *p) {
71     // Berechnet die Summe, die Differenz und das Produkt von a und b
72     // und bringt sie in die Variablen *s, *d bzw. *p.
73     *s = a + b;
74     *d = a - b;
75     *p = a * b;
76  } // berechne
77  ...
78  void main() {
79     int n1=5, n2=3, summ, diff, prod;
80     berechne(n1, n2, &summ, &diff, &prod);
81     ...
```

Eine Funktion kann nur **ein** Ergebnis **liefern** (mit der return-Anweisung). Mit Hilfe von Parametern eines Adresstyps kann eine Funktion ihren Aufrufern aber beliebig viele weitere Ergebnisse (nicht liefern aber) **zur Verfügung stellen**. Nach dem Aufruf der Funktion `berechne` in Zeile 30 stehen dem Aufrufer in den Variablen `summ`, `diff` und `prod` **drei** Ergebnisse zur Verfügung.

Die Variablen `n1` und `n2` werden "ganz normal" **per Wert** an die Funktion `berechne` übergeben. Die Variablen `summ`, `diff` und `prod` werden dagegen **per Adresse** übergeben (indem ihre Adressen **per Wert** übergeben werden).

Zu jedem Typ T gibt es 4 Adresstypen, deren Werte und Variablen "mit mehr oder weniger Schreibberechtigungen verbunden sind".

Beispiel 7: Die 4 Adresstypen des Typs `int`

```
82                       // Welche Variablen sind unveraenderbar (konstant)?
83                       // *an  | an     (n = 1, 2, 3, 4)
84                       //------|------
85       int *       a1; // nein | nein
86 const int *       a2; // ja   | nein
87       int * const a3; // nein | ja
88 const int * const a4; // ja   | ja
```

Die Adressvariable `a2` vom Typ `const int *` (Adresse-von-int-Konstante) kann die Adresse einer `int`-Variablen oder einer `int`-Konstanten enthalten, aber sie berechtigt in keinem Fall zum verändern ihres Zieles. Entsprechendes gilt auch für `a4`. Die Adressvariable `a3` vom Typ `int * const` (konstante-Adresse-von-int-Variable) darf selbst nicht verändert werden, erlaubt aber das Verändern ihrer Zielvariablen `*a3`.

Große Variablen (z.B. Reihungen und Verbunde) übergibt man häufig per Adresse, um ein zeitaufwendiges Kopieren der Variablen zu vermeiden. Wenn man deutlich machen möchte, dass die Variable in der aufgerufenen Funktion nicht verändert wird, sollte man den formalen Parameter mit einem Typ wie `const typ *` ("Adresse einer unveränderbaren Variablen des Typs typ", kurz: Adresse einer typ-Konstanten) vereinbaren. Allerdings gibt es Tricks, wie ein Programmierer die Wirkung der `const`-Angabe außer Kraft setzen kann.

Damit die Namen von Adresstypen nicht zu einfach zu lesen und zu verstehen sind, darf man anstelle von `const typ` überall auch `typ const` schreiben.

Konkrete Anwendungen der 4 Adresstypen des Typs `int` (`int*`, `const int*`, `int* const` und `const int * const`) findet man im Beispielprogramm `Zeiger05`.

Der Adressoperator `&` und der Zieloperator `*` heben sich gegenseitig auf. Ist `v` irgendeine Variable, dann kann man statt `v` auch `*&v` schreiben (das Ziel der Adresse von `v` ist `v`). Ist `av` irgendeine Adressvariable (die nicht gerade den Wert `NULL` enthält), dann kann man statt `av` auch `&*av` schreiben (die Adresse des Ziels von `av` ist `av`).

Achtung: Nach der folgenden Mehrfachvereinbarung

```
89 int * d1, d2, * d3, d4;
```

gehören die Variablen `d1` und `d3` zum Typ `int *`, die Variablen `d2` und `d4` dagegen zum Typ `int` gehören (und nicht zum Typ `int *` !)..

Das folgende Beispiel soll zeigen, wie man Adressen von Funktionen in entsprechenden Adressvariablen speichern und wie man das Ziel einer solchen Adressvariablen aufrufen kann:

Beispiel 8: Adressvariablen für Adressen von **Funktionen**

```
 90 // Datei Zeiger06.c
 91 /* ------------------------------------------------------------------------
 92 In C haben auch Funktionen Adressen, die man in Variablen speichern kann.
 93 ------------------------------------------------------------------------ */
 94 void gibAus1(int n) {
 95    printf("gibAus1: n hat den Wert   %d\n", n);
 96 } // gibAus1
 97
 98 void gibAus2(int n) {
 99    printf("gibAus2: n has the value  %d\n", n);
100 } // gibAus1
101
102 void gibAus3(int n) {
103    printf("gibAus3: n tiene el valor %d\n", n);
104 } // gibAus1
105
106 // Namen fuer einen Funktionstyp und einen Adresstyp vereinbaren:
107 typedef void ft1(int);
108 typedef ft1* aft1;
109
110 // Zum Typ ft1 gehoeren alle Funktionen mit dem Rueckgabetyp void und
111 // einem int-Parameter (z.B. die Funktionen gibAus1 bis gibAus3).
112 // Zum Typ aft1 gehoeren der Wert NULL und alle Adressen von Funktionen
113 // des Typs ft1 (z.B. die Adressen &gibAus1 bis &gibAus3).
114
115 // Alternativ haette man den Adresstyp aft1 auch "direkt" (ohne den
116 // Hilfstyp ft1) wie folgt vereinbaren koennen:
117 // typedef void (*aft1) (int);
118
119 aft1 funk   = gibAus1; // oder gleichbedeutend: ... = &gibAus1;
120 aft1 tab[3] = {gibAus1, gibAus2, gibAus3};
121 // ------------------------------------------------------------------------
122 int main() {
123    int i;
124    printf("Zeiger06: Jetzt geht es los!\n");
125    printf("--------------------------\n");
126    funk(5);
127    funk = gibAus2;
128    funk(3);
129    printf("--------------------------\n");
130    for (i=0; i<3; i++) {
131       tab[i](3*i + 2);
132    }
133    printf("--------------------------\n");
134    return 0;
135 } // main
136 /* ------------------------------------------------------------------------
137 Ausgabe des Programms Zeiger06:
138
139 Zeiger06: Jetzt geht es los!
140 --------------------------
141 gibAus1: n hat den Wert   5
142 gibAus2: n has the value  3
143 --------------------------
144 gibAus1: n hat den Wert   2
145 gibAus2: n has the value  5
146 gibAus3: n tiene el valor 8
147 --------------------------
148 ------------------------------------------------------------------------ */
```

16 Reihungen (arrays) und Adressen

Beispiel 1: Ein Programm mit int-Reihungen

```
 1 // Datei Reihungen01.c
 2 /* ------------------------------------------------------------------------
 3 Reihungen (arrays) mit int-Komponenten werden vereinbart und ausgegeben.
 4 Die Wirkung des Operators sizeof wird demonstriert.
 5
 6 Merke: In C "weiss eine Reihung nicht, wie lang sie ist"!
 7 --------------------------------------------------------------------- */
 8 #include <stdio.h>
 9
10 #define LENA 5 // Laenge der Reihung ra
11 #define LENB 3 // Laenge der Reihung rb
12
13 int ra[LENA] = {+3, -5, +7, +2, -4};
14 int rb[LENB] = {-2, +9, -8};
15 // ------------------------------------------
16 void gibAus(int rf[], int laenge) {
17    // Gibt die Reihung rf in lesbarer Form aus
18    int i;
19    printf("E sizeof(rf): %d Bytes\n", sizeof(rf));
20    printf("F %d Kompos  : ", laenge);
21    for (i=0; i<laenge; i++) {
22       printf("%d ", rf[i]);
23    }
24    printf("\n");
25 } // gibAus
26 // ------------------------------------------
27 int main(void) {
28    printf("Reihungen01: Jetzt geht es los!\n");
29    printf("-------------------------------------\n");
30    printf("A sizeof(ra)              : %2d Bytes\n", sizeof(ra));
31    printf("B sizeof(rb)              : %2d Bytes\n", sizeof(rb));
32    printf("C sizeof(ra)/sizeof(ra[0]): %2d Kompos\n",
33              sizeof(ra)/sizeof(ra[0]));
34    printf("D sizeof(rb)/sizeof(rb[0]): %2d Kompos\n",
35              sizeof(rb)/sizeof(rb[0]));
36    printf("-------------------------------------\n");
37    gibAus(ra, LENA);
38    printf("-------------------------------------\n");
39    gibAus(rb, LENB);
40    printf("-------------------------------------\n");
41    gibAus(rb, LENA);                                  // Oooops!
42    printf("-------------------------------------\n");
43    return 0;
44 } // main
45 // ------------------------------------------
46 /* ------------------------------------------------------------------------
47 Ausgabe des Programms Reihungen01:
48
49 Reihungen01: Jetzt geht es los!
50 -------------------------------------
51 A sizeof(ra)              : 20 Bytes
52 B sizeof(rb)              : 12 Bytes
53 C sizeof(ra)/sizeof(ra[0]):  5 Kompos
54 D sizeof(rb)/sizeof(rb[0]):  3 Kompos
55 -------------------------------------
56 E sizeof(rf): 4 Bytes
57 F 5 Kompos  : 3 -5 7 2 -4
58 -------------------------------------
59 E sizeof(rf): 4 Bytes
```

```
60 F 3 Kompos  : -2 9 -8
61 ------------------------------------
62 E sizeof(rf): 4 Bytes
63 F 5 Kompos  : -2 9 -8 168035352 0
64 ------------------------------------
65 ----------------------------------------------------------------------- */
```

Einige Eigenschaften von Reihungen in C sind für einen Java-Programmierer möglicherweise ziemlich verwunderlich.

1. Eine Reihung in einem C-Programm enthält **keine** Information darüber, wie **lang** sie ist.

2. Einer **Funktion**, die Reihungen unterschiedlicher Länge bearbeiten soll (siehe z.B. die Funktion `gibAus`, definiert ab Zeile 16), muss man in aller Regel die Länge der zu bearbeitenden Reihung als **zusätzlichen Parameter** übergeben.

3. Greift man mit einem Ausdruck wie etwa `r[i]` auf die i-te Komponente einer Reihung r zu, so prüft der Ausführer **nicht**, ob der Index i "einen vernünftigen Wert" hat. Der Programmierer kann dem Ausführer also z.B. befehlen, auf die 7. Komponente einer Reihung der Länge 5 zuzugreifen. Was bei einem solchen Zugriff passiert, hängt von der Implementierung und möglicherweise von zahlreichen anderen Umständen (Tageszeit, Wetter etc.) ab. In Zeile 41 wird dem Ausführer befohlen, 5 Komponenten der Reihung `rb` auszugeben, obwohl `rb` nur 3 Komponenten hat. Eine entsprechende Ausgabe (erzeugt unter Windows bei mildem Sonnenschein) ist in Zeile 63 wiedergegeben.

4. Die Definition der Reihungsvariablen `ra` (in Zeile 13) bewirkt, dass eine Reihung mit `LEN` vielen int-Komponenten erzeugt und initialisiert wird. Der Name `ra` kann danach wie eine Variable des Typs `int * const` (konstante Adresse von `int`) verwendet werden. Das Ziel der Adresskonstanten `ra` ist die erste Komponente der Reihung (die mit dem Index 0 und dem Anfangswert +3). Der Wert von `ra` kann nicht verändert werden (da `ra` eine Konstante ist), aber die Komponenten `ra[0]`, `ra[1]`, ... der Reihung sind `int`-Variablen, auf die man lesend und schreibend zugreifen darf.

5. Wendet man den Operator `sizeof` auf eine Reihungsvariable an, so erhält man manchmal die Länge der Reihung und manchmal nur die Länge einer Adresse (häufig: 4 Bytes). In den Zeilen 30 bis 35 sind die Definitionen der Reihungen `ra` und `rb` sichtbar und man erhält dort die wirkliche Länge der Reihungen. In Zeile 14 "weiß der Ausführer nicht", wie lang die Reihung ist, deren Adresse in `rf` steht. Deshalb liefert `sizeof` dort nur die Länge der Adresse `rf` (4 Bytes), nicht die Länge der Reihung.

Das Programm `Reihungen02` stimmt weitgehend mit `Reihungen01` überein und enthält nur eine etwas andere Funktion `gibAus`:

Beispiel 2: Eine andere Funktion `gibAus` und ihre Ausgabe

```
66 void gibAus(char * name, int * const rf, int len) {
67    // Verlaesst sich darauf, dass rf die Adresse einer int-Reihung der
68    // Laenge len enthaelt. Gibt jede Komponente dieser Reihung und ihre
69    // Adresse dreimal aus (mit unterchiedlichen aber gleichbedeutenden
70    // Notationen):
71
72    int i;
73    printf("Die Reihung %s hat %d Komponenten:", name, len);
74    for (i=0; i<len; i++) {
75       printf("\n%s[%d]: ", name, i);
76       printf("%2d, ", rf[i]  );     // Notation 1: Reihungsnotation
77       printf("%2d, ", *(rf+i));     // Notation 2: Adressnotation
78       printf("%2d, ", i[rf]  );     // Notation 3: Verdrehte Reihungsnot.
79       printf("Adresse: ");
80       printf("%p, ", &rf[i]  );     // Notation 1
81       printf("%p, ", &*(rf+i));     // Notation 2
82       printf("%p. ", &i[rf]  );     // Notation 3
```

```
 83     }
 84     printf("\n");
 85  } // gibAus
 86  ...
 87  /* ---------------------------------------------------------------------------
 88  Ausgabe des Programms Reihungen02:
 89
 90  Reihungen02: Jetzt geht es los!
 91  ---------------------------------------------------------
 92  Die Reihung ra hat 5 Komponenten:
 93  ra[0]:  3,  3,  3, Adresse: 0x403010, 0x403010, 0x403010.
 94  ra[1]: -5, -5, -5, Adresse: 0x403014, 0x403014, 0x403014.
 95  ra[2]:  7,  7,  7, Adresse: 0x403018, 0x403018, 0x403018.
 96  ra[3]:  2,  2,  2, Adresse: 0x40301c, 0x40301c, 0x40301c.
 97  ra[4]: -4, -4, -4, Adresse: 0x403020, 0x403020, 0x403020.
 98  ---------------------------------------------------------
 99  Die Reihung rb hat 3 Komponenten:
100  rb[0]: -2, -2, -2, Adresse: 0x403024, 0x403024, 0x403024.
101  rb[1]:  9,  9,  9, Adresse: 0x403028, 0x403028, 0x403028.
102  rb[2]: -8, -8, -8, Adresse: 0x40302c, 0x40302c, 0x40302c.
103  ---------------------------------------------------------
```

Die Zeilen 76 bis 78 und 80 bis 82 sollen deutlich machen, dass die Notationen `rf[i]`, `*(rf+i)` und (merkwürdigerweise auch) `i[rf]` gleichbedeutend sind. Alle drei Notationen bezeichnen die i-te Komponente der Reihung, deren Anfangsadresse in `rf` steht.

Das folgende Beispiel soll noch einmal gezeigt werden, wie man mit Adressen **rechnen** kann, um z.B. auf die Komponenten einer Reihung zuzugreifen. Insbesondere soll deutlich gemacht werden was es bedeutet, eine Adressvariable um 1 zu erhöhen.

Beispiel 3: Adressvariablen um 1 erhöhen

```
104  // Datei Zeiger02.c
105  /* ------------------------------------------------------------------------
106  Erhoeht man eine Adresse-von-short-Variable um 1, so wird sie in
107  Wirklichkeit um sizeof(short) erhoeht (so dass sie auf die naechste
108  short-Variable zeigt).
109
110  Erhoeht man eine Adresse-von-double-Variable um 1, so wird sie in
111  Wirklichkeit um sizeof(double) erhoeht (so dass sie auf die naechste
112  double-Variable zeigt).
113  ------------------------------------------------------------------------ */
114  #include <stdio.h>   // E/A in Bytestroeme, printf, sprintf, ...
115  #define LEN 5
116  // ------------------------------------------------------------------------
117  short  sr01[LEN] = {111, 222, 333, 444, 555};
118  double dr01[LEN] = {1.1, 2.2, 3.3, 4.4, 5.5};
119  int    i;          // Index fuer for-Schleifen
120  // ------------------------------------------------------------------------
121  void gibAusSR(short sr[]) {
122     // Gibt die Adressen und Werte der Reihungskomponenten sr[i] aus.
123     short * avs = &sr[0]; // Adresse von short
124
125     for (i=0; i<LEN; i++) {
126        printf("avs: %p, i: %d, avs+i: %p, *(avs+i): %d\n",
127                avs,     i,     avs+i,     *(avs+i));
128     }
129  } // gibAusSR
130  // ------------------------------------------------------------------------
131  void gibAusDR(double dr[]) {
132     // Gibt die Adressen und Werte der Reihungskomponenten dr[i] aus.
133     double * avd = &dr[0]; // Adresse von double
134
```

```
135     for (i=0; i<LEN; i++) {
136        printf("avd: %p, i: %d, avd+i: %p, *(avd+i): %3.1f\n",
137                avd,     i,     avd+i,     *(avd+i));
138     }
139  } // gibAusDR
140  // ------------------------------------------------------------------------------
141  int main() {
142     printf("Zeiger02: Jetzt geht es los!\n");
143     printf("----------------------------------------------------------\n");
144     printf("sizeof(short) : %d Bytes\n", sizeof(short));
145     printf("sizeof(double): %d Bytes\n", sizeof(double));
146     printf("----------------------------------------------------------\n");
147     gibAusSR(sr01);
148     printf("----------------------------------------------------------\n");
149     gibAusDR(dr01);
150     printf("----------------------------------------------------------\n");
151
152     return 0;
153  } // main
154  /* ------------------------------------------------------------------------------
155  Ausgabe des Programms Zeiger02:
156
157  Zeiger02: Jetzt geht es los!
158  ----------------------------------------------------
159  sizeof(short) : 2 Bytes
160  sizeof(double): 8 Bytes
161  ----------------------------------------------------
162  avs: 0040A128, i: 0, avs+i: 0040A128, *(avs+i): 111
163  avs: 0040A128, i: 1, avs+i: 0040A12A, *(avs+i): 222
164  avs: 0040A128, i: 2, avs+i: 0040A12C, *(avs+i): 333
165  avs: 0040A128, i: 3, avs+i: 0040A12E, *(avs+i): 444
166  avs: 0040A128, i: 4, avs+i: 0040A130, *(avs+i): 555
167  ----------------------------------------------------
168  avd: 0040A134, i: 0, avd+i: 0040A134, *(avd+i): 1.1
169  avd: 0040A134, i: 1, avd+i: 0040A13C, *(avd+i): 2.2
170  avd: 0040A134, i: 2, avd+i: 0040A144, *(avd+i): 3.3
171  avd: 0040A134, i: 3, avd+i: 0040A14C, *(avd+i): 4.4
172  avd: 0040A134, i: 4, avd+i: 0040A154, *(avd+i): 5.5
173  ----------------------------------------------------
174  ---------------------------------------------------------------------------- */
```

17 Den Cursor positionieren und Zeichen ohne Return einlesen

Wenn der Benutzer einem Programm ohne Grabo (grafische Benutzeroberfläche, engl. GUI, graphical user interface) Daten über eine Tastatur eingeben will, dann muss er seine Eingabe normalerweise mit der Return-Taste abschließen. Beim Eingeben eines einzigen Zeichens stellt der Druck auf die Return-Taste einen Zusatzaufwand von 100% dar. Verschiedene Betriebssysteme bieten einen speziellen Befehl an, mit dem ein Zeichen sofort eingelesen wird, d.h. sobald der Benutzer auf eine Taste gedrückt hat und bevor er auf eine weitere Taste (z.B. die Return-Taste) drückt. Mit einem solchen Befehl kann man sehr "reaktive" und benutzerfreundliche Programme realisieren.

Mit den Ein-/Ausgabebefehlen der C-Standardbibliothek kann man u.a. Daten zum Bildschirm ausgeben, aber man kann damit nicht den Bildschirmzeiger (cursor) ausdrücklich zu einer bestimmten Stelle auf dem Bildschirm bewegen. Verschiedene Betriebssystem bieten einen speziellen Befehl an, mit dem man den Bildschirmzeiger positionieren kann.

Auch Windows-Betriebssysteme enthalten Funktionen zum Lesen von Zeichen ohne Return, zum Positionieren des Bildschirmzeigers, zum Festlegen und Ändern der Farben in einem Dos-Fenster etc. Der Modul `Konsole` soll es C-Programmierern erleichtern, diese Funktionen aufzurufen.

Beispiel 1: Die Datei `Konsole.h`

```
 1 // Datei Konsole.h
 2 /* ------------------------------------------------------------------------
 3 Der Modul Konsole besteht aus den Dateien Konsole.h und Konsole.c.
 4 Mit diesem Modul (und einem geeignet C-Compiler) kann man unter einem
 5 Windows-Betriebssystem
 6 - das Standardausgabefenster loeschen (ClearScreen)
 7 - fuer das Standardausgabefenster Farben festlegen (SetColors)
 8 - einzelne Zeichen und ganze Zeichenketten zu bestimmten Stellen des
 9   Standardausgabefensters ausgeben (Put)
10 - im Standardausgabefenster den Cursor positionieren (GotoXY)
11 - von der Standardeingabe einzelne Zeichen "sofort" und ohne Echo lesen,
12   (d.h. der Benutzer muss nicht auf die Return-Taste druecken und das
13   eingegebene Zeichen erscheint nicht auf dem Bildschirm (GetImmediate
14   und GetImmediateAny).
15 - die Groesse des Standardausgabefensters veraendern
16   (FenstergroesseMaximal und Fenstergroesse_80_25)
17 - zu einer Farbe zyklisch die nachfolgende Farbe bzw. die vorhergehende
18   Farbe berechnen lassen (ColorSucc und ColorPred).
19 - sich den Namen einer Farbe (als C-String) liefern lassen (ColorName).
20
21 Die bool-Funktionen liefern TRUE, wenn "alles geklappt hat". Sie liefern
22 FALSE, wenn ein Fehler auftrat (z.B. weil die Parameter falsch waren).
23 ------------------------------------------------------------------------ */
24 #ifndef Konsole_h
25 #define Konsole_h
26
27 #include <windows.h> // fuer den Typ BOOL
28 //------------------------------------------------------------------------
29 // Diese Funktion muss einmal und vor allen anderen Funktionen dieses
30 // Moduls aufgerufen werden:
31 void initKonsole(void);
32
33 // Groesse des Standardausgabefensters:
34 short AnzZeilen (void); // Liefert die Anzahl der Zeilen
35 short AnzSpalten(void); // Liefert die Anzahl der Spalten
36 //------------------------------------------------------------------------
37 // Eine Farbe ist eine 4-Bit-Binaerzahl, mit je einem Bit fuer RED, GREEN
38 // BLUE und BRIGHT. BRIGHT plus RED ist BRIGHT_RED, BLUE plus GREEN ist
39 // CYAN, BRIGHT plus BLUE plus GREEN ist BRIGHT_CYAN etc. Die Farbe
40 // bright black wird hier als GRAY bezeichnet:
```

```
41 enum Color {
42    BLACK     = 0, BLUE           = 1, GREEN        = 2, CYAN        = 3,
43    RED       = 4, MAGENTA        = 5, YELLOW       = 6, WHITE       = 7,
44    GRAY      = 8, BRIGHT_BLUE    = 9, BRIGHT_GREEN =10, BRIGHT_CYAN =11,
45    BRIGHT_RED=12, BRIGHT_MAGENTA=13, BRIGHT_YELLOW=14, BRIGHT_WHITE=15
46 };
47
48 #define COLOR enum Color
49 //------------------------------------------------------------------------
50 BOOL ClearScreen (const COLOR  vordergrund,
51                   const COLOR  hintergrund);
52 // Loescht den Bildschirm und schickt den Bildschirmzeiger (cursor) nach
53 // Hause (Spalte 0, Zeile 0). Wahlweise kann eine Farbe fuer den Vorder-
54 // grund und eine Farbe fuer den Hintergrund angegeben werden. Diese
55 // Farben gelten dann fuer alle Zellen des Standardausgabefensters und
56 // alle zukuenftigen Ausgaben zum Standardausgabefenster (bis zum naechsten
57 // Aufruf dieser Funktion ClearScreen oder einer SetColors-Funktion).
58 //------------------------------------------------------------------------
59 BOOL SetColorsA  (const COLOR  vordergrund,
60                   const COLOR  hintergrund);
61 // Legt die Farben fuer *zukuenftige* Ausgaben zum Standardausgabefenster
62 // fuer *alle* Zeichen-Zellen fest. Die Farben gelten nur fuer Ausgaben wie
63 // z.B. printf("Hallo!");, nicht aber fuer Ausgaben mit den Funktionen
64 // PutC und PutS, die in diesem Modul vereinbart werden (siehe unten).
65 // Die Farben bleiben wirksam bis zum naechsten Aufruf einer SetColors-
66 // Funktion oder der ClearScreen-Funktion. Ausgaben mit printf erscheinen
67 // moeglicherweise erst nach einem Aufruf wie fflush(stdout);
68 //------------------------------------------------------------------------
69 BOOL SetColorsE  (const short  spalte,
70                   const short  zeile,
71                   const COLOR  vordergrund,
72                   const COLOR  hintergrund);
73 // Legt die Hintergrundfarbe und die Vordergrundfarbe fuer *eine* Zeichen-
74 // Zelle des Standardausgabefensters fest (das Standardausgabefenster
75 // besteht aus AnzZeilen * AnzSpalten vielen Zeichen-Zellen).
76 //------------------------------------------------------------------------
77 BOOL SetColorsV  (const short  von_spalte,
78                   const short  bis_spalte,
79                   const short  von_zeile,
80                   const short  bis_zeile,
81                   const COLOR  vordergrund,
82                   const COLOR  hintergrund);
83 // Legt die Hintergrundfarbe und die Vordergrundfarbe fuer ein Rechteck
84 // auf dem Bildschirm (fuer *viele* Zeichen-Zellen) fest. Das Rechteck
85 // kann aus beliebig vielen Zeichen-Zellen bestehen (maximal aus allen
86 // Zellen des Bildschirms).
87 //------------------------------------------------------------------------
88 BOOL PutS        (const short  spalte,
89                   const short  zeile,
90                   const char  *string);
91 // Gibt den string zum Standardausgabefenster aus. Das erste Zeichen des
92 // Textes wird in die Zelle (spalte, zeile) geschrieben.
93 // ACHTUNG: Die Position des Bildschirmzeigers (cursor) wird durch dieses
94 // Unterprogramm *nicht* veraendert!
95 //------------------------------------------------------------------------
96 BOOL PutC        (const short  spalte,
97                   const short  zeile,
98                   const char   zeichen);
99 // Gibt das zeichen zum Standardausgabefenster aus, an die Position (spalte,
100 // zeile).
101 // ACHTUNG: Die Position des Bildschirmzeigers (cursor) wird durch dieses
102 // Unterprogramm *nicht* veraendert!
```

```
103 //------------------------------------------------------------------------
104 BOOL GotoXY      (const short  spalte,
105                   const short  zeile);
106 // Bringt den Bildschirmzeiger im Standardausgabefenster an die Position
107 // (spalte, zeile). Nachfolgende Ausgaben (z.B. puts("Hallo!";) erfolgen
108 // ab dieser neuen Position. Ausgaben mit printf erscheinen evtl. erst
109 // nach einem Aufruf wie fflush(stdout); .
110 //------------------------------------------------------------------------
111 BOOL GetImmediateAny(char  *zeichen,
112                      short *psVirtualKeyCode,
113                      int   *piControlKeyState);
114 // Liest ein Zeichen von der Standardeingabe (Tastatur). Der eingebende
115 // Benutzer muss seine Eingabe *nicht* mit der Return-Taste abschliessen.
116 // Die Eingabe erfolgt *ohne Echo* (d.h. das eingegebene Zeichen erscheint
117 // nicht automatisch auf dem Bildschirm). Mit diesem Unterprogramm kann
118 // *jeder* Tastendruck ("any key") gelesen werden (auch ein Druck auf eine
119 // Taste wie Pfeil-nach-oben, Bild-nach-unten, Umschalt-Taste, linke
120 // Alt-Taste, Rechte Alt-Taste, linke Strg-Taste, rechte Strg-Taste oder
121 // auf eine Funktionstaste F1, F2 etc. etc.). Die folgenden Funktionen
122 // sollen das Analysieren des Parameters *piControlKeyState erleichtern.
123 // -----------------------------------------------------------------------
124 // Funktionen zur Ananlyse des Parameters *piControlKeyState der
125 // Funktion GetImmediateAny:
126 int Capslock_On        (int iControlKeyState);
127 int Enhanced_Key       (int iControlKeyState);
128 int Left_Alt_Pressed   (int iControlKeyState);
129 int Left_Ctrl_Pressed  (int iControlKeyState);
130 int Numlock_On         (int iControlKeyState);
131 int Right_Alt_Pressed  (int iControlKeyState);
132 int Right_Ctrl_Pressed(int iControlKeyState);
133 int Scrolllock_On      (int iControlKeyState);
134 int Shift_Pressed      (int iControlKeyState);
135 //------------------------------------------------------------------------
136 BOOL GetImmediate(char  *zeichen);
137 // Liest ein Zeichen von der Standardeingabe (Tastatur). Der eingebende
138 // Benutzer muss seine Eingabe *nicht* mit der Return-Taste abschliessen.
139 // Die Eingabe erfolgt *ohne Echo* (d.h. das eingegebene Zeichen erscheint
140 // nicht automatisch auf dem Bildschirm). Mit diesem Unterprogramm koennen
141 // nur "normale Zeichen" (Buchstaben, Ziffern, Sonderzeichen etc.) einge-
142 // lesen werden. Steuerzeichen (z.B. Pfeil-nach-oben, Bild-nach-unten etc.)
143 // werden vom Betriebssystem "verschluckt".
144 //------------------------------------------------------------------------
145 BOOL FenstergroesseMaximal();
146 // Vergroessert das Standardausgabefenster auf die maximale Groesse (die
147 // von der Groesse des Bildschirms und dem verwendeten Font abhaengt).
148 //------------------------------------------------------------------------
149 BOOL Fenstergroesse_80_25();
150 // Veraendert die Groesse des Standardausgabefensters so, dass es 25 Zeilen
151 // mit je 80 Spalten enthaelt.
152 //------------------------------------------------------------------------
153 COLOR ColorSucc(const COLOR c);
154 // Zyklische Nachfolgerfunktion fuer Color (nach BRIGHT_WHITE kommt BLACK)
155 COLOR ColorPred(const COLOR c);
156 // Zyklische Vorgaengerfunktion fuer Color (vor BLACK liegt BRIGHT_WHITE)
157 const char *ColorName(const COLOR c);
158 // Liefert den Namen der Farbe c als String.
159 //------------------------------------------------------------------------
160 #endif // Konsole_h
```

Beispiel 2: Ein kleines Programm, in dem der Modul `Konsole` benutzt wird

```
161 // Datei KonsoleKeys.c
```

```
162 /* ------------------------------------------------------------------
163 Liest mit der Funktion GetImmediateAll() Zeichen von der Standardeingabe
164 (d.h. die Zeichen werden ohne Echo gelesen und ohne auf Return zu
165 warten) und gibt folgende Daten zur aktuellen Ausgabe aus:
166 - Das gelesene Zeichen (bzw. ein Blank, falls es nicht druckbar ist)
167 - Das gelesene Zeichen als Zahl (zwischen 0 und 255)
168 - Den (geraeteunabhaengigen) virtual key code des Zeichens
169 - Den control key status
170 - Die einzelnen Bits des control key status (Capslock_On, Enhanced_Key
171   bis Shift_Pressed)
172 ------------------------------------------------------------------ */
173 #include "Konsole.h" // fuer ClearScreen, GetImmediateAny, GotoXY, ...
174 #include <stdio.h>
175
176 int main() {
177
178    int           alles_ok;
179    char          zeichen_c  = 'x'; // Anfangswert ungleich 'q'
180    unsigned char zeichen_uc;
181    int           zeichen_i;
182    short         vkc;              // virtual key code
183    int           cks;              // control key state
184
185    #define              LEN  10
186    char alsChar        [LEN];
187    char alsInt         [LEN];
188    char virtualKeyCode [LEN];
189    char controlKeyState[LEN];
190    char alles_ok_s     [LEN];
191
192    char * capslock_on        ;
193    char * enhanced_key       ;
194    char * left_alt_pressed   ;
195    char * left_ctrl_pressed  ;
196    char * numlock_on         ;
197    char * right_alt_pressed  ;
198    char * right_ctrl_pressed ;
199    char * scrolllock_on      ;
200    char * shift_pressed      ;
201    // ---------------------------------------------------------------
202    initKonsole();
203    ClearScreen(BRIGHT_WHITE, BLACK);
204
205    printf("Programm KonsoleKeys: Jetzt geht es los!\n");
206    printf("Bitte Tasten druecken (q zum Beenden): \n");
207
208    while (zeichen_c != 'q') {
209       alles_ok   = GetImmediateAny(&zeichen_c, &vkc, &cks);
210       zeichen_uc = (unsigned char)(zeichen_c);
211       zeichen_i  = (int)          (zeichen_uc);
212
213       // Nicht-druckbare Zeichen durch Punkt '.' ersetzen:
214       if (zeichen_uc < 32) zeichen_c = '.';
215
216       // Informationen ueber die gedrueckte(n) Tase(n) in Strings umwandeln:
217
218       capslock_on        = Capslock_On       (cks) ? "on " : "off";
219       enhanced_key       = Enhanced_Key      (cks) ? "on " : "off";
220       left_alt_pressed   = Left_Alt_Pressed  (cks) ? "on " : "off";
221       left_ctrl_pressed  = Left_Ctrl_Pressed (cks) ? "on " : "off";
222       numlock_on         = Numlock_On        (cks) ? "on " : "off";
223       right_alt_pressed  = Right_Alt_Pressed (cks) ? "on " : "off";
224       right_ctrl_pressed = Right_Ctrl_Pressed(cks) ? "on " : "off";
```

```
225        scrolllock_on      = Scrolllock_On     (cks) ? "on " : "off";
226        shift_pressed      = Shift_Pressed     (cks) ? "on " : "off";
227
228 //     sprintf(alsChar,          "'%c'    ", zeichen_i);
229 //     sprintf(alsChar,          "'%c'    ", zeichen_uc);
230        sprintf(alsChar,          "'%c'    ", zeichen_c);
231        sprintf(alsInt,           "%d      ", zeichen_i);
232        sprintf(virtualKeyCode,   "%hd     ", vkc);
233        sprintf(controlKeyState, "0x%04x ", cks);
234        sprintf(alles_ok_s,       "%d      ", alles_ok);
235
236        // Informationen ueber die gedrueckte(n) Taste(n) ausgeben:
237        PutS(0,  2, "Zeichen als char  : "); PutS(21,  2, alsChar);
238        PutS(0,  3, "Zeichen als int   : "); PutS(21,  3, alsInt);
239        PutS(0,  4, "VirtualKeyCode    : "); PutS(21,  4, virtualKeyCode);
240        PutS(0,  5, "ControlKeyState   : "); PutS(21,  5, controlKeyState);
241        PutS(0,  6, "CapsLock          : "); PutS(21,  6, capslock_on);
242        PutS(0,  7, "Enhanced_Key      : "); PutS(21,  7, enhanced_key);
243        PutS(0,  8, "Left_Alt_Pressed  : "); PutS(21,  8, left_alt_pressed);
244        PutS(0,  9, "Left_Ctrl_Pressed : "); PutS(21,  9, left_ctrl_pressed);
245        PutS(0, 10, "Numlock_On        : "); PutS(21, 10, numlock_on);
246        PutS(0, 11, "Right_Alt_Pressed : "); PutS(21, 11, right_alt_pressed);
247        PutS(0, 12, "Right_Ctrl_Pressed: "); PutS(21, 12, right_ctrl_pressed);
248        PutS(0, 13, "Scrolllock_On     : "); PutS(21, 13, scrolllock_on);
249        PutS(0, 14, "Shift_Pressed     : "); PutS(21, 14, shift_pressed);
250        PutS(0, 15, "Alles O.K.        : "); PutS(21, 15, alles_ok_s);
251     } // while
252
253     ClearScreen(WHITE, BLACK);
254     printf("Programm KonsoleKeys: Das war's erstmal!\n");
255
256     return 0;
257 } // main
258 // ------------------------------------------------------------------------
```

Dieses Programm läuft natürlich nur unter einem **Windows**-Betriebssystem. Unter **Linux** oder **Solaris** müsste man die dort vorhandenen Funktionen zum sofortigen Lesen von Zeichen, Positionieren des Cursors etc. aufrufen.

18 Anhang: Compilieren und Binden ohne und mit make

Angenommen, ein C-Programm besteht aus den Quelldateien dat01.c und dat02.c und den Kopfdateien dat01.h und dat02.h. Daraus soll eine ausführbare Datei namens dat.exe erzeugt werden.

Beispiel 1: Mit dem Gnu-C-Compiler gcc

```
 1 Alle Schritte auf einmal:
 2 gcc -o dat.exe dat01.c dat02.c
 3
 4 In Einzelschritten:
 5 gcc -c dat01.c                     // dat01.c zu dat01.o compilieren
 6 gcc -c dat02.c                     // dat02.c zu dat02.o compilieren
 7 gcc -o dat.exe dat01.o dat02.o     // zu dat.exe binden
 8
 9 gcc-Optionen anzeigen lassen:
10 gcc --help | more
11 gcc --target-help | more
12
13 Mehr Infos zu gcc:
14 man gcc
```

Wenn man in Zeile 2 oder 7 die Option -o dat.exe weg lässt, heißt die ausführbare Datei a.exe. Statt dat.exe kann man auch nur dat (ohne .exe) schreiben.

Beispiel 2: Mit dem C-Compiler lcc-win32:

```
15 Alle Schritte auf einmal:
16 Geht nur mit make (siehe unten)!
17
18 In Einzelschritten:
19 lcc dat01.c                          // dat01.c zu dat01.obj compilieren
20 lcc dat02.c                          // dat02.c zu dat02.obj compilieren
21 lcclnk -o dat.exe dat01.obj dat02.obj // zu dat.exe binden
22
23 lcc-Optionen anzeigen lassen:
24 lcc | more
25
26 lcclnk-Optionen anzeigen lassen:
27 lcclnk | more
```

Wenn man in Zeile 21 die Option -o dat.exe weg lässt, heißt die ausführbare Datei dat01.exe. (nach der zuerst angegebenen .obj-Datei). Gibt man anstelle von dat.exe nur dat an, so heißt die ausführbare Datei auch nur dat (und kann deshalb nicht ausgeführt werden!).

Beispiel 3: Mit dem Borland-C/C++-Compiler bcc32:

```
28 Alle Schritte auf einmal:
29 bcc32 -edat.exe dat01.c dat02.c
30
31 In Einzelschritten:
32 bcc32 -c dat01.c                   // dat01.c zu dat01.obj compilieren
33 bcc32 -c dat02.c                   // dat02.c zu dat02.obj compilieren
34 bcc32 -edat.exe dat01.o dat02.o    // zu dat.exe binden
35
36 bcc32-Optionen anzeigen lassen:
37 bcc32
```

Wenn man einem Java-Ausführer die Hauptklasse eines Programms übergibt ("die Klasse mit der main-Methode"), dann findet er alle anderen Klassen des Programms in aller Regel automatisch und ohne weitere Hilfe des Programmierers.

C-Ausführer haben diese Fähigkeit **nicht**. Welche Dateien zu einem Programm gehören, wie sie zusammenhängen und in welchen Einzelschritten sie kompiliert und dann zusammengebunden werden sollen, beschreibt man deshalb üblicherweise in so genannten **Makedateien** und lässt diese vom Programm `make` ausführen. Verändert man nur eine oder wenige Dateien des Programms und lässt dann die Makedatei erneut ausführen, so kann `make` automatisch feststellen, welche Dateien erneut zu compilieren sind und welche anderen Schritte durchgeführt werden müssen. In diesem Fall führt make keine unnötigen Schritte durch. Natürlich funktioniert das nur dann, wenn man das C-Programm und die einzelnen durchzuführenden Schritte in der Makedatei "richtig" beschrieben hat.

Bei Java ist sozusagen "ein Teil des make-Programms schon in den Java-Ausführer eingebaut" und der Programmierer braucht keine Makedateien zu schreiben. Mit Makedateien kann man andererseits mehr erreichen als nur mit einem Java-Ausführer, z.B. ein Programm erzeugen, dessen Teile in verschiedenen Sprachen geschrieben sind, zwischendurch Dateien kopieren und löschen, Partitionen formatieren und vieles mehr.

Auch unter Windows kann man die UNIX-Werkzeuge (utilities) `make`, `gcc`, `rm`, `man`, `ls` etc. etc. leicht installieren (z.B. die kostenlose Distribution Cygwin der Firma Cygnus). Im Folgenden wird vorausgesetzt, dass diese Werkzeuge zur Verfügung stehen.

Beispiel 4: Eine Makedatei zum Erzeugen der Datei `dat.exe` aus den Dateien `dat01.c`, `dat02.c`, `dat01.h` und `dat02.h`

```
38 # Datei dat.mfg (Makedatei fuer den Gnu-C-Compiler gcc)
39 # Alle eingerueckten Zeilen MUESSEN MIT EINEM TABZEICHEN beginnen
40 # und duerfen NICHT mit Blanks ("Leerzeichen") beginnen!!!
41
42 dat.exe: dat01.o dat02.o
43   gcc -o dat.exe dat01.o dat02.o # Zeile beginnt mit Tabzeichen!
44   dat.exe
45
46 dat01.o: dat01.c dat01.h dat02.h
47   gcc -c dat01.c                 # Zeile beginnt mit Tabzeichen!
48
49 dat02.o: dat02.c dat01.h dat02.h
50   gcc -c dat02.c                 # Zeile beginnt mit Tabzeichen!
51
52 clean:
53   rm -f *.o                      # Zeile beginnt mit Tabzeichen!
54   rm -f *.exe                    # Zeile beginnt mit Tabzeichen!
```

Wenn man das Programm `make` ohne Parameter aufruft, sucht es im aktuellen Arbeitsverzeichnis nach einer Makedatei namens `makefile` oder `Makefile` (unter UNIX sind das zwei unterschiedliche Dateinamen, unter Windows gelten die Namen als gleich) und führt die Makebefehle in dieser Datei aus. Soll eine Makedatei mit einem anderen Namen (z.B. `dat.mfg`) ausgeführt werden, muss man `make` mit der Option `-f` (wie file) etwa wie folgt aufrufen:

```
55 make -f dat.mfg
```

Achtung: In obiger Makedatei muss auch die Zeile 44 mit einem **Tabzeichen** beginnen, aber ein entsprechender Kommentar am Ende dieser Zeile würde eine Fehlermeldung des Programms `dat.exe` auslösen.

Verstößt man gegen die Tabzeichen-Regel (indem man z.B. am Anfang von Zeile 53 ein paar Blanks eintippt), gibt das `make`-Programm eine Fehlermeldung etwa wie die folgende aus:

```
56 dat.mfg:53: *** missing separator.  Stop.
```

Mit dem Aufruf `make -h` kann man sich alle Optionen des Programms `make` anzeigen lassen, mit dem Aufruf `man make` "die kleine Dokumentation" zu `make` (manual pages, ca. 150 Zeilen) und mit dem Aufruf `info make` "die große Dokumentation" (ca. 10 000 Zeilen). Wer keine Lust hat zu lernen,

in einer man-Dokumentation oder in einer info-Dokumentation zu navigieren, kann diese Dokumentationen auch in Textdateien umlenken (mit dem Befehl man make > dat1.txt bzw. info man > dat2.text) und die Textdateien mit einem guten Editor lesen.

Anmerkung: Die Festlegung, dass bestimmte Zeilen in einer Makedatei mit einem **Tabzeichen** beginnen müssen, ist ein ganz heißer Kandidat für die Auszeichnung als unsinnigste Entwurfsentscheidungen aller Zeiten.

Erläuterungen zu der Makedatei dat.mfg:

Diese Datei enthält 4 **Ziele** (goals) namens dat.exe, dat01.o, dat02.o und clean. Die Namen der ersten drei Ziele sind Dateinamen, der Name des vierten Ziels (clean) ist "frei erfunden" und kein Dateiname.

In Zeile 42 steht, dass das Ziel dat.exe von den Zielen dat01.o und dat02.o **abhängt**. In den Zeilen 43 und 44 stehen die Befehle, mit denen das Ziel dat.exe **realisiert** (oder: erreicht) werden soll. Diese Befehle werden von make aber nur ausgeführt ("das Ziel dat.exe wird nur dann realisiert"), wenn mind. eine der Dateien dat01.o und dat02.o **jünger** ist, als die Datei dat.exe (oder wenn noch keine Datei dat.exe existiert).

Ganz entsprechend hängt das Ziel dat01.o (siehe Zeile 46) von den Dateien dat01.c, dat01.h und dat02.h ab. Wenn eine dieser Dateien jünger ist als die Datei dat01.o wird das Ziel dat01.o realisiert, indem der Befehl in Zeile 47 ausgeführt wird (d.h. die Datei dat01.c wird compiliert).

Für das Ziel dat02.o (siehe Zeile 49 bis 50) gilt ganz Entsprechendes wie für dat01.o.

Das erste Ziel dat.exe hängt vom zweiten und dritten Ziel (dat01.o und dat02.o) ab. Das vierte Ziel (clean) hängt nicht mit den anderen drei Zielen zusammen.

Normalerweise sorgt make dafür, dass das **erste** Ziel in der Makedatei realisiert wird (falls es noch nicht auf dem neusten Stand ist). Um dieses erste Ziel zu realisieren müssen zuerst die Ziele, von denen es abhängt, geprüft und eventuell realisiert werden (und für diese Ziele gilt das Gleiche etc.).

Will man, dass ein anderes als das **erste** Ziel realisiert wird, muss man es beim Aufruf von make ausdrücklich angeben, etwa so:

```
57 make -f dat.mfg clean
```

Jetzt realisiert make nur das Ziel clean, d.h. mit dem remove-Befehl rm werden alle .o- und alle .exe-Dateien gelöscht, siehe Zeile 53 und 54. Ohne die Option -f wie "force" würde der erste rm-Befehl mit einer Fehlermeldung abgebrochen, falls es keine .o-Dateien zu löschen gibt, und der zweite rm-Befehl würde in einem solchen Fall **nicht** ausgeführt. Mit der -f-Option werden immer beide Befehle ausgeführt und produzieren keine lästigen Fehlermeldungen wenn sie nichts zu tun haben.

Ein Ziel wie clean, dessen Name **kein Dateiname** ist, wird wie eine "unendlich alte" Datei behandelt (falls es geprüft wird, wird es auch realisiert).

Hängt ein Ziel von einer Datei ab, die kein Ziel ist (wie etwa das Ziel dat01.o von der nicht-Ziel-Datei dat01.h) dann gilt: Wenn die Datei existiert, wird ihr Alter geprüft, wenn sie nicht existiert bricht make ab mit einer Fehlermeldung wie der folgenden:

```
58 make: *** No rule to make target `dat01.h', needed by `dat01.o'.  Stop.
```

Eine der wichtigsten Aufgaben des make-Programms ist es, unnötige Schritte (z.B. Compilations- und Bindeschritte) **nicht** auszuführen. Lässt man die obige Makedatei zweimal nacheinander ausführen (so wie in Zeile 55, ohne Zielangabe), dann erscheint spätestens beim zweiten Mal folgende Fehlermeldung:

```
59 make: `dat.exe' is up to date.
```

In diesem Fall hat make festgestellt, dass die Datei dat.exe älter ist als die Dateien dat01.o und dat02.o ("nach der letzten Erzeugung von dat.exe wurden dat01.o und dat02.o nicht mehr geändert") und dass dat01.o älter ist als dat01.c , dat01.h und dat02.h ("nach der letzten Erzeugung von dat01.o wurden dat01.c, dat01.h und dat02.h nicht mehr geändert") und dass dat02.o älter ist als dat01.c, dat01.h und dat02.h. Daraus hat make geschlossen, dass kein Ziel erneut realisiert werden muss und hat "nichts gemacht". Eigentlich sollte man die Meldung in Zeile 59 als eine **Erfolgsmeldung** interpretieren.

Das Ziel clean ist nie auf dem neusten Stand (up to date) und wird bei jedem Aufruf (wie in Zeile 57) erneut realisiert (d.h. die beiden rm-Befehle in Zeile 53 und 54 werden erneut ausgeführt, auch wenn dadurch keine Dateien mehr gelöscht werden).

Beispiel 5: Eine etwas andere Makedatei zur Erzeugung der Datei dat.exe:

```
60 # Datei dat.mfb (Makedatei fuer den Borland-C/C++-Compiler bcc32)
61
62 dat.exe: dat01.obj dat02.obj dat01.h dat02.h
63   bcc32 -edat.exe dat01.obj dat02.obj
64   dat.exe
65
66 dat01.obj: dat01.c dat01.h dat02.h
67   bcc32 -c dat01.c
68
69 dat02.obj: dat02.c dat01.h dat02.h
70   bcc32 -c dat02.c
71
72 clean:
73   rm -f *.obj
74   rm -f *.tds
75   rm -f *.exe
```

In dieser Makedatei wird das Programm dat.exe mit Hilfe des Borland-Compilers-und-Binders bcc32 erzeugt. Dieser Compiler erzeugt mehr Hilfsdateien als der gcc, deshalb sind für das Ziel clean mehr rm-Befehle nötig als im vorigen Beispiel.

Aufgabe 1: Schreiben Sie eine Makedatei namens dat.mfl, die das Programm dat.exe mit Hilfe des Compilers lcc und des Binders lcclnk erzeugt.

Aufgabe 2: Angenommen, wir hätten vergessen, in Zeile 66 die Datei dat02.h anzugeben. Welche Konsequenzen könnte dieses Versehen haben? Was passiert, wenn wir in diesem Fall die Datei dat02.h ändern und danach das Programm dat.exe erneut erzeugen lassen (natürlich mit der Makedatei dat.mfb)?

19 Anhang: Die Compiler gcc und lcc

Zur Zeit (September 2003) gibt es vermutlich keinen kostenlosen C-Compiler, der den C99-Standard schon voll erfüllt. Aber der Gnu-C-Compiler **gcc** und der C-Compiler **lcc-win32** erfüllen den Standard schon weitgehend und werden laufend in diese Richtung weiterentwickelt.

Der **gcc** (Gnu Compiler Collection) ist nicht nur ein **C-Compiler**, sondern eine (nicht-grafische) Benutzeroberfläche (a front end) für **mehrere** Compiler (für die Sprachen **Fortran**, **Pascal**, **Modula**, **Ada**, **treelang**, **Objective-C**, **C++**, **Java** und **C**). Speziell der Gnu-C-Compiler ist vermutlich der auf die meisten Plattformen portierte C-Compiler, den es heute gibt, und steht insbesondere auf vielen Spezialprozessoren zur Verfügung. Der Gnu-C-Compiler "kennt" nicht nur **eine** Sprache C, sondern kann mit der Option `std=...` auf alle wichtigen Varianten von C "umgeschaltet und eingeschränkt" werden, etwa so:

```
> gcc -c -std=gnu89 dat.c    // ANSI-C (oder ISO C89) plus Gnu-Erweiterungen
```

Anstelle von **gnu89** kann man auch angeben: **C89** (ISO C89 ohne Gnu-Erweiterungen), **gnu99** (ISO C99 plus Gnu-Erweiterungen, C99 noch nicht ganz vollständig), **C99** (ISO C99 ohne Gnu-Erweiterungen) und weitere Standard-Varianten. Zur Zeit ist **gnu89** der Standardwert (default), soll aber in naher Zukunft durch **gnu99** abgelöst werden (aktuelle Informationen dazu findet man unter `http://gcc.gnu.org/onlinedocs/gcc-3.3.1/gcc/Standards.html`).

gcj und **gij** sind zwei neuere und interessante Ergänzungen zu der Compilersammlung **gcc**. Mit dem Compiler **gcj** kann man **Java**-Programme wahlweise in Bytecode (.class-Dateien) oder in Maschinenprogramme (unter Windows: in `.exe`-Dateien) übersetzen, und das unter Linux, Windows, Solaris und anderen Betriebssystemen. Mit dem Interpreter **gij** kann man `.class`-Dateien ausführen lassen (ähnlich wie mit dem Interpreter **java** von Sun). Die folgende Makedatei soll einen Eindruck davon vermitteln, was man mit den Werkzeugen **gcj** und **gij** machen kann:

```
 1 # Datei makefile
 2 # ------------------------------------------------------------------------
 3 # Von den drei Quelldateien Hall10.java, Hallo11.java und Hallo12.java
 4 # enthalten zwei eine main-Methode (Hallo10 und Hallo12). Die Klassen
 5 # Hallo10, Hallo11 und Hallo12 bilden zusammen ein Programm, Hallo12 ist
 6 # auch allein ein vollstaendiges Programm.
 7 #
 8 # Ziele dieser Makedatei:
 9 # 1. Hallo12.java wird in eine .class-Datei uebersetzt und
10 #    mit gij    ausgefuehrt.
11 # 2. Hallo12.java wird in eine   .exe-Datei uebersetzt und
12 #    ("direkt") ausgefuehrt.
13 # 3. Hallo10.java, Hallo11.java und Hallo12.java werde in
14 #    .class-Dateien uebersetzt und Hallo10.class wird mit gij ausgefuehrt.
15 # 4. Hallo10.java, Hallo11.java und Hallo12.java werden in
16 #    eine Datei Hallo.exe ueberstzt und ("direkt") ausgefuehrt.
17 # 5. clean loescht (wie immer) alle mit gcj erzeugten Dateien.
18 # ------------------------------------------------------------------------
19 # Im gcj-Befehl kann man die Quelldateien in beliebiger Reihenfolge
20 # angeben.
21 #
22 # Uebersetzt man Quelldateien (mit der Option -C) in .class-Dateien,
23 # dann entscheidet sich erst beim Ausfuehren mit gij, von welcher Klasse
24 # die main-Methode ausgefuehrt werden soll. Nach gij gibt man nur die
25 # Hauptklasse eines Programms ("die, deren main-Methode ausgefuehrt werden
26 # soll") an. Die Nebenklassen werden automatisch geladen wenn sie zum
27 # ersten Mal gebraucht werden.
28 #
29 # Die Option -Wa,-W wird von gcj an den Assembler as weitergereicht und
30 # verhindert, dass der as Warnungen ausgibt.
31 #
32 # Die Option --main=... legt (beim Uebersetzen in eine .exe-Datei) die
33 # Hauptklasse fest.
```

```
34 # ------------------------------------------------------------------------
35 Alle4: Hallo12.class Hallo12.exe Hallo10.class Hallo.exe
36
37 Hallo12.class: Hallo12.java
38   gcj -C Hallo12.java
39   gij    Hallo12
40
41 Hallo12.exe: Hallo12.java
42   gcj -o Hallo12.exe -Wa,-W --main=Hallo12 Hallo12.java
43   Hallo12.exe
44
45 Hallo10.class: Hallo10.java Hallo11.java Hallo12.java
46   gcj -C Hallo12.java Hallo10.java Hallo11.java
47   gij    Hallo10
48
49 Hallo.exe: Hallo10.java Hallo11.java Hallo12.java
50   gcj -o Hallo.exe -Wa,-W --main=Hallo10 Hallo10.java Hallo11.java Hallo12.java
51   Hallo.exe
52
53 clean:
54   rm -f *.class
55   rm -f *.exe
56   rm -f *.o
```

Der **gcj** "kennt schon" fast alle Standardklassen der Java-Version 1.3.1, mit Ausnahme der **awt**- und **swing**-Klassen. Es werden noch Mitarbeiter gesucht, die bei der Implementierung dieser Klassen helfen. Mit dem **gcj** und dem **gcc** ist es relativ leicht, ein Programm zum Teil in **Java** und zum Teil in **C++** (oder C) zu schreiben.

Der Compiler **lcc-win32** ist ein reiner C-Compiler (kein "gebremster C++-Compiler"!), compiliert sehr schnell und realisiert bereits einen großen Teil des C99-Standards. Er ist ganz auf Windows-Betriebssysteme spezialisiert und steht auf anderen Plattformen nicht zur Verfügung. Ergänzt wird er durch eine integrierte Entwicklungsumgebung (`wedit`) mit guten Hilfefunktionen, einem brauchbaren integrierten Editor, verschiedenen Assistentenprogrammen (wizards), Bibliotheken und zahlreichen Erweiterungen der Sprache C. Die nicht-kommerzielle Benutzung ist kostenlos. Weitere Infos findet man u.a. unter der Adresse `http://www.cs.virginia.edu/~lcc-win32/`.

20 Anhang: Kurzübersicht über Standard-Kopfdateien

Die Standard-Kopfdateien enthalten (Präprozessor-) Makros mit Parametern (z.B. `assert`), Makros ohne Parameter (`INT_MAX`, `FLT_MAX`, `true`, `false`), Deklarationen von Funktionen (z.B. `isalpha`, `isdigit`, `sin`, `cos`), Vereinbarungen von Typnamen und Typen (z.B. `size_t`, `ptrdiff_t`, `int8_t`, `intmax_t`) und Definitionen von Variablen (z.B. `errno`).

```
 1 <assert.h>   // Makro assert.
 2 <ctype.h>    // isalpha, isdigit, islower, isupper, tolower, ...
 3 <errno.h>    // errno, EDOM, ERANGE, ...
 4 <float.h>    // FLT_MAX, FLT_MIN, DBL_MAX, DBL_MIN, ...
 5 <inttypes.h>// PRId8, PRIdLEAST8, PRIdFAST8, ...
 6 <iso646.h>   // and, or, xor, bitand, bitor, compl, not, ...
 7 <limits.h>   // INT_MIN, INT_MAX, LONG_MIN, LONG_MAX, ...
 8 <math.h>     // HUGE_VAL, sin, cos, tan, log, pow, sqrt, ...
 9 <signal.h>   // Signale senden/fangen, signal, raise, SIG_DFL, SIG_IGN,
10              // SIGABRT, SIGFPE, SIGILL, SIGINT, SIGSEGV, SIGTERM, ...
11 <setjmp.h>   // Lange Rueckspruenge, jump_buf, setjmp, longjmp.
12 <stdarg.h>   // Ermoeglicht variable Anzahl von Parametern
13 <stdbool.h>  // Makros bool, true, false.
14 <stddef.h>   // Typen ptrdiff_t, size_t, wchar_t, Konstante NULL
15 <stdint.h>   // int8_t, uint8_t, int_least8_t, uint_least8_t,
16              // int_fast8_t, intptr_t, uintptr_t, intmax_t, ...
17 <stdio.h>    // E/A in Bytestroeme, printf, scanf, sprintf, ...
18 <stdlib.h>   // atof, atoi, atol, rand, malloc, calloc, free, exit,
19              // abs, div, getenv, system, bsearch, qsort, ...
20 <string.h>   // memcopy, memmove, strcpy, strcat, ...
21 <time.h>     // tm_sec, tm_min, tm_hour, clock, time, ...
```

21 Anhang: Der Editor vi

Von diesem Editor gibt es zahlreiche Varianten (unter Namen wie z.B. **elvis**, **nvi**, **vim** etc.). Auf jedem UNIX-System gibt es mindestens eine Variante und in bestimmten Situationen ist ein **vi** der einzige Editor, der zur Verfügung steht. Wenn man an moderne Editoren wie **TextPad**, **Ultraedit** oder **Kate** gewöhnt ist, wird man vermutlich Mühe haben, sich auf den **vi** umzustellen. Aber nach wenigen Monaten des Übens findet man ihn nicht mehr so schlimm wie am Anfang (und nach ein paar Jahren ist der vi der mit Abstand beste Editor, den man sich vorstellen kann :-).

21.1 Den vi Starten

```
vi dat.c        // Die Datei dat.c mit dem vi öffnen
vi -R dat.c     // Nur zum Lesen öffnen, Veränderungen sind nicht möglich
```

Falls noch keine Datei namens **dat.c** existiert, wird eine solche angelegt. Nach dem Starten befindet der **vi** sich im **Befehlsmodus** (visual command mode, siehe unten).

21.2 Veränderungen an Dateien abspeichern und vi beenden

```
:w             // Editierte Datei abspeichern, vi nicht beenden
ZZ             // Alle geöffneten Dateien abspeichern und vi beenden
:q             // vi beenden ("quit"), wenn man keine Dateien verändert hat
:q!            // Alle Veränderungen an Dateien verwerfen und vi beenden
```

21.3 Hilfe

```
:help (oder :h) // In einem Teil des Bildschirms wird eine Hilfedatei angezeigt
                // (die man natürlich nur lesen aber nicht verändern kann).
                // Mit /Karl/ kann man darin nach dem Wort Karl suchen, mit
                // :q kann man die Hilfedatei schliessen.
:h quit         // Zeigt Hilfe zum quit-Kommando an
:h help         // Zeigt Hilfe zum help-Kommando an
:h z*           // Zeigt Hilfe zu allen Kommandos an, die mit z beginnen
```

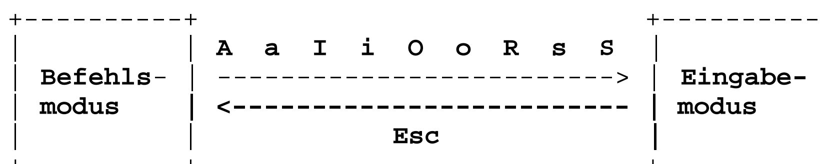
21.4 Den vi-Modus wechseln

Der **vi** befindet sich immer in einem **Modus**, entweder im **Befehlsmodus** (visual command mode) oder im **Eingabemodus**. Im **Befehlsmodus** werden die Eingaben des Benutzers als Befehle interpretiert, nicht als Daten. Nur im **Eingabemodus** kann man Daten in die Datei einfügen.

Nach dem Starten befindet der **vi** sich im **Befehlsmodus**. Vom **Befehlsmodus** gelangt man durch eine der folgenden Befehle in den **Eingabemodus**: A, a, I, i, O, o, R, s, S. Der Buchstabe bestimmt, wo der neue Text eingefügt wird:

i (bzw. **a**): vor (bzw. nach) dem aktuellen Zeichen
I (bzw. **A**): am Anfang (bzw. am Ende) der aktuellen Zeile
O (bzw. **o**): in eine neue Zeile vor (bzw. nach) der aktuellen Zeile.
R (wie "replace"): Ab dem aktuellen Zeichen werden die Zeichen der aktuellen Zeile überschrieben.
s (wie "substitute"): Das aktuelle Zeichen wird durch die Eingabe ersetzt.
S (wie "substitute"): Die aktuelle Zeile   wird durch die Eingabe ersetzt.

Mit **Esc** gelangt man vom **Eingabemodus** zurück in den **Befehlsmodus**. Wenn man nicht sicher ist, ob der **vi** sich im **Befehls-** oder **Eingabemodus** befindet, sollte man **Esc** drücken. Danach befindet man sich garantiert im **Befehlsmodus**.



Im Befehlsmodus kann man Doppelpunkt-Befehle und andere Befehle eingeben.

21.5 Wichtige Doppelpunkt-Befehle

```
:set ai            // autoindent, automatisches Einrücken ("wie vorige Zeile")
:set nu            // Zeilen-Nrn auf dem Bildschirm anzeigen (nicht abspeichern)
:17                // Zur Zeile 17 springen
:$                 // Zur letzten Zeile springen
```

21.6 Suchen und Ersetzen

```
/Otto/             // In den folgenden Zeilen ("nach unten") nach Otto suchen
?Otto?             // In den vorhergehenden Zeilen ("nach oben") nach Otto suchen
/                  // Letzten Suchbefehl nach unten wiederholen
?                  // Letzten Suchbefehl nach oben  wiederholen
:s/Otto/Emil/      // In der aktuellen Zeile das erste Vorkommen von Otto durch
                   // Emil ersetzen
:&                 // Den letzten Ersetzungsbefehl wiederholen
:1;$s/Otto/Emil/   // Auf jeder Zeile das erste Vokommen von Otto durch Emil ers.
:1;$s/Otto/Emil/g  // Auf jeder Zeile alle Vorkommen von Otto durch Emil ers.
:1;$s/Otto/Emil/gc // Ebenso, aber vor jeder Ersetzung fragen ("confirm")
:5;8s/Otto/Emil/gc // Ebenso, aber nur von Zeile 5 bis Zeile 8
```

21.7 undo und redo

```
:u         // Den letzten Befehl rückgängig machen (bei vi nur einmal!)
:red       // Den rückgängig gemachten Befehl wieder ausführen (nicht bei vi!)
```

21.8 Kopieren, Löschen und Wiedereinsetzen

```
dd         // Lösche  die aktuelle Zeile (in den yank-Puffer)
yy         // Kopiere die aktuelle Zeile (in den yank-Puffer)
3dd        // Lösche  3 Zeilen (in den yank-Puffer)
3yy        // Kopiere 3 Zeilen (in den yank-Puffer)
p          // Kopiere den yank-Puffer hinter die aktuelle Zeile/Cursorposition
P          // Kopiere den yank-Puffer vor    die aktuelle Zeile/Cursorposition
x          // Lösche  ein Zeichen (in den yank-Puffer)
3x         // Lösche   3  Zeichen (in den yank-Puffer)
yw         // Kopiere ein Wort    (in den yank-Puffer)
3yw        // Kopiere  3  Worte   (in den yank-Puffer)
dw         // Lösche  ein Wort    (in den yank-Puffer)
3dw        // Lösche   3  Worte   (in den yank-Puffer)
```

21.9 Einrücken und Ausrücken

```
>>         // Die aktuelle Zeile um eine Stufe einrücken
<<         // Die aktuelle Zeile um eine Stufe ausrücken
```

21.10 Eine vi-Konfigurationsdatei .exrc in das HOME-Verzeichnis schreiben

Beim Start sucht der **vi** im HOME-Verzeichnis des Benuters eine Datei namens **.exrc**. Falls er eine findet, führt er die Befehle darin aus. Der Befehl **cd** (ohne Parameter) wechselt ins HOME-Verzeichnis des Benutzers. Der Befehl **pwd** gibt das akutelle Verzeichnis aus (so kann man herausfinden, wo das HOME-Verzeichnis des Benutzers liegt). Die Datei **.exrc** kann z.B. folgende Befehle enthalten (siehe oben 5.):

```
set autoindent        // oder: set ai
set number            // oder: set nu
set shiftwidth=3      // oder: set sw=3, legt Länge einer Einrückstufe fest
set tabstop=3         // oder: set ts=3, legt Länge einer Tab-Stufe    fest
```

21.11 Absturz des vi

Während der **vi** eine Datei namens **dat.c** bearbeitet, speichert er wichtige Informationen in einer sog. swap-Datei namens **dat.c.swp**. Sollte der **vi** abstürzen, hat man zumindest eine Chance, mit Hilfe der swap-Datei die Datei **dat.c** wiederherzustellen. Einfach erneut **vi dat.c** aufrufen und den Anweisungen folgen (oder gleich **vi -r dat.c** aufrufen).

21.12 Ausblick

Es gibt noch **sehr viele** weitere **vi**-Befehle. Insbesondere kann man mit dem **vi** mehrere Dateien gleichzeitig bearbeiten (z.B. zerstören :-) und dabei mehrere Puffer benützen (nicht nur den **einen yank**-Puffer). Einige davon werden im folgenden Abschnitt beschrieben.

22 Anhang: Der Editor vim und mehr vi(m)-Befehle

22.1 Hilfe und Tutorial zur vi-Varianten vim

Zum `vim` gibt es mehr als 20 000 Zeilen `help`-Dateien und ein Tutorial, mit dem man die wichtigsten Grundbefehle lernen und üben kann. Das Tutorial ruft man in einem Cygwin-Fenster so auf:

```
> vimtutorial     // englische Version
> vimtutorial es  // spanische Version, man kann auch franzoesisch, italienisch
                  // russisch und japanisch angeben, eine deutsche Version
                  // gibt es noch nicht.
```

22.2 Eine Initialisierungsdatei (.vimrc) bereitlegen

Nachdem man den `vim` gestartet hat, kann man ihn mit dem folgenden Befehl nach seiner **Versions**-Nr und den **Optionen** fragen, mit denen er compiliert wurde:

```
:ve
```

Am Ende der Ausgabe verrät der `vim`, in welcher Reihenfolge er wo nach **Init(ialisierungs)dateien** mit welchen **Namen** sucht. Die Befehle in der zuerst gefunden systemweiten Initdatei und danach die in der zuerst gefundenen Benutzer-Initdatei führt er aus:

1.1. Systemweite Initdatei:	`%VIM%\vimrc`	`// vimrc ohne Punkt!`
1.2. Systemweite Initdatei:	`.../usr/share/vim/vimrc`	`// vimrc ohne Punkt!`
2.1. Benutzer-Initdatei (vim):	`%HOME%\.vimrc`	`// vimrc mit Punkt!`
2.2. Benutzer-Initdatei (ex):	`%HOME%\.exrc`	`// exrc mit Punkt!`

Dabei ist `%VIM%` der Inhalt der Umgebungsvariablen `VIM` (`%HOME%` entsprechend). In den Initdateien können z.B. folgende Befehle stehen:

```
 1 " --------------------------------------------
 2 " Datei .vimrc (oder vimrc oder .exrc)
 3 " Kommandos zum Initialisieren des Editors vim
 4 " --------------------------------------------
 5
 6 set nocompatible        " vim soll sich anders als vi verhalten
 7 syntax on               " Der Typ einer Datei wird automatisch festgestellt
 8                         " (z.B. C-Quelldatei, Ada-Quelldatei, Perl etc.)
 9                         " und der Inhalt entsprechend "bunt dargestellt"
10                         " (z.B. Literale rot, Kommentare blau, ...)
11 set number              " Zeilen-Nrn anzeigen
12 set shiftwidth=3        " Eine Einrueckstufe: 3 Zeichen
13 set tabstop=3           " Ein Tabzeichen:     3 Zeichen
14 set cpoptions-=u        " Beliebig viele undo-Befehle (nicht nur einer
15                         " wie bei v, compatibility option "not u")
16 set ruler               " Zeilen-Nr und Spalten-Nr anzeigen
17 set showcmd             " Unvollstaendig Kommandos anzeigen
18 set incsearch           " Inkrementelles Suchen: Waehrend man den Such-
19                         " begriff eintippt, springt der Cursor schon "zur
20                         " ersten passenden Stelle".
21
22 if has("autocmd")       " Nur wenn autocommands "eincompiliert" wurden:
23                         " Dateitypen automaisch erkennen und richtig
24                         " einruecken:
25   filetype plugin indent on
26                         " Fuer alle Text-Dateien Textbreite gleich 78:
27   autocmd FileType text setlocal textwidth=78
28 endif " has("autocmd")
```

Mit der Option `-u` kann der Benutzer beim Starten des `vim` auch eine (alle anderen ausschliessende) Initdatei angeben, etwa so:

```
> vim -u c:\meinVerzeichnis\meineVimrc hallo01.c
```

22.3 Weitere Befehle im Befehlsmodus

```
.              // (Ein Punkt) Wiederholt das letzte Einfuege- oder Loesch-Kommando.
ga             // Gibt das aktuelle Zeichen in dezimaler, oktaler und hexadezimaler
               // Darstellung aus (siehe auch Semikolonbefehl :as)
%              // Wenn das aktuelle Zeichen oder Wort Teil eines Paares ist,
               // springt der Cursor zum "Partner" des Zeichens bzw. Wortes.
               // Zeichen-Paare: '('und ')', '[' und ']', '{' und '}'.
               // Wort-Paare: #ifdef und #endif, #if, #else und #endif.
5iabc<esc>     // Fuege 5 Mal "abc" vor dem aktuellen Zeichen ein
5Aabc<esc>     // Fuege 5 Mal "abc" am Ende der aktuellen Zeile ein
J (gross!)     // Mache aus der aktuellen und der naechsten Zeile EINE Zeile
2J             // Gleiche Wirkung wie J
3J             // Mache aus der aktuellen und den 2 naechsten Zeilen EINE Zeile
```

22.4 Ein paar Befehle im Eimgabemodus (^ bedeutet <Strg>)

```
^T             // Aktuelle Zeile eine Stufe mehr einrücken (nach rechts)
^D             // Aktuelle Zeile eine Stufe mehr ausrücken (nach links)
^N             // Vervollstaendige das aktuelle Wort ("Vorbilder vorher")
^P             // Vervollstaendige das aktuelle Wort ("Vorbilder nachher")
```

22.5 Weitere Semikolon-Befehle

```
:as            // Gibt das aktuelle Zeichen in dezimaler, oktaler und hexadezimaler
               // Darstellung aus (siehe auch Befehlsmodus-Befehl ga)
:l             // (ell, nicht eins!) Gibt die akutelle Zeile aus und stellt dabei
               // Steuerzeichen in lesbarer Form dar (newline-Zeichen als $,
               // Tabzeichen als ^I etc.)
:l3            // 3 Zeilen      mit l ausgeben
:4,7l          // Zeile 4 bis 7 mit l ausgeben
:!ls -la       // Fuehrt das Kommando "ls -la" aus und kehrt dann zum vim zurueck.
               // Anstelle von "ls -la" kann man ein beliebiges Kommando angeben.
:↑             // (Preil rauf)   voriger   Semikolon-Befehl aus der "Geschichte"
:↓             // /Pfeil runter) naechster Semikolon-Befehl aus der "Geschichte"
:mak           // Fuehre das Kommando make (ohne Parameter) aus. Falls dabei
               // (z.B. von einem Compiler oder einem Binder) ein Fehler erkannt
               // wird, dann springe anschliessend zu der entsprechenden Stelle
               // im Quellprogramm.
:mak clean     // Fuehre das Kommando make mit Parameter clean aus, sonst wie mak.
```

22.6 Mehrere Dateien "gleichzeitig" editieren

Den `vim` mit einem Befehl wie `vim dat1.c dat2.txt dat3.c` starten. Mit den Semikolon-Befehlen `:next` und `:prev` kommt man zur jeweils nächsten bzw. vorigen Datei.

22.7 Eine Datei hexadezimal anzeigen lassen und editieren

```
vim -b dat.c // Die Datei mit der Option -b ("binary") oeffnen
:%!xxd         // Die Datei wird hexadezimal und zeichenweise dargestellt.
               // Aendert man eine Darstellung, wird die andere NICHT ent-
               // sprechend geaendert. Beim Aendern der hexadezimalen Darstellung
               // darf man keine Zeichen einfuegen oder entfernen, am besten nur
               // mit r oder R Zeichen ersetzen.
:%!xxd -r      // Die hexadezimale Darstellung wird zurueck in eine "normale",
               // zeichenweise Darstellung umgewandelt.
```

23 Das Programm ctags

Mit dem Programm ctags (von Darren Hiebert) kann man sich das "hin und her Springen" zwischen mehreren C-Quelldateien (während man sie mit dem Editor vi oder vim bearbeitet) erleichtern.

Angenommen, alle Quelldateien `q01.c`, `q02.c`, ... eines C-Programms liegen in einem Verzeichnis `v`. Man mache `v` zum aktuellen Arbeitsverzeichnis und rufe das Programm `ctags` auf (siehe dazu den Hinweis weiter unten). Dadurch wird im Verzeichnis `v` eine Datei namens `tags` erstellt, die von vielen C-Größen Informationen darüber enthält, in welcher Quelldatei und Zeile sie definiert werden. Zu diesen Größen gehören: Variablen, Funktionen, (Präprozessor-) Makros, `enum`-, `struct`- und `union`-Typen, `enum`-Konstanten und `struct`-Komponenten.

Hinweis: Man sollte das Programm `ctags` auch unter Windows nur in einem `bash`-Fenster ("Cygwin-Fenster") aufrufen, nicht in einer Dos-Eingabeaufforderung (dort gibt `ctags` schwer verständliche Fehlermeldungen aus und funktioniert manchmal nicht richtig).

```
$ ctags *          // Erstelle im aktuellen Arbeitsverzeichnis eine Datei namens
                   // tags mit Informationen ueber alle Dateien in diesem Ver-
                   // zeichnis (deren Dateinamenerweiterung, z.B. .c oder .h etc.,
                   // dem Programm ctags "bekannt sind").
$ ctags -R         // Beruecksichtige zusaetzlich auch alle Dateien in allen Unter-
                   // zeichnisse des aktuellen Verzeichnisses (R wie rekursiv).
```

Nachdem man die Datei `tags` erstellt hat, stehen einem folgende zusätzlichen `vi`-Befehle (auch im `vim`) zur Verfügung:

```
$ vi -t otto        // Oeffne die Datei, in der otto definiert wird, und positio-
                    // niere den Cursor am Anfang der Definition.
:tag otto           // Springe zu der Datei und Zeile, in der otto definiert wird.
<Strg>-+            // Springe zu der Datei und Zeile, in der der aktuelle
                    // Bezeichner (der momentan unter dem Cursor liegt) definiert
                    // wird.
<Strg>-T            // Springe zurueck zum Ausgangspunkt des letzten tags-Sprunges
                    // (nach mehreren tags-Spruengen kann man auch mehrmals
                    // zurueckspringen).
```

Die folgenden Befehle sind nützlich, wenn in den Quelldateien z.B. **mehrere** Variablen namens `otto` definiert werden oder eine Funktion und eine Variable namens `emil` oder **mehrere** Makros, Funktionen und Variablen namens `anna` etc.

```
:ts otto            // Zeige alle Definitionen von Groessen namens otto. Durch
                    // Eingabe einer entsprechenden Nr kann man danach zu einer
                    // dieser Definitionen (Datei und Zeile) springen.
:tn                 // Springe zur naechsten Definition mit gleichem Bezeichner
:tp                 // Springe zur vorigen   Definition mit gleichem Bezeichner
:t5                 // Springe zur 5.        Definition mit gleichem Bezeichner
:tr                 // Springe zur ersten    Definition mit gleichem Bezeichner
:tl                 // Springe zur letzten   Definition mit gleichem Bezeichner
```

Hinweis: Mit dem Programm `ctags` kann man auch Quelldateien anderer Sprachen als C (C++, `make`, `Fortran`, `Java`, `Eiffel`, `PHP`, ...) bearbeiten und die von `ctags` erstellte `tags`-Datei kann auch von anderen Editoren als `vi` (`emacs`, `nedit`, ...) verwendet werden.