Allgemeine Informatik III - Teil 1: Die Programmiersprache C

Skript: F. Schweiggert und M. Grabert

16. Oktober 2000

Fakultät Mathematik u. Wirtschaftswissenschaften
Abteilung Angewandte Informationsverarbeitung
Vorlesungsbegleiter (gültig ab WS 2000/2001)

Anmerkungen:

- Auf eine detaillierte Unterscheidung zwischen *BSD Unix* und *System V Unix* wird hier verzichtet.
- Die enthaltenen Beispiel-Programme wurden zum großen Teil unter Linux entwickelt und sind weitestgehend unter Solaris getestet (fur konstruktive Hinweise ist der Autor dankbar).
- Die Beispiele sollen jeweils gewisse Aspekte verdeutlichen und erheben nicht den Anspruch von Robustheit und Zuverlässigkeit. Man kann alles anders und besser machen.
- Details zu den behandelten / verwendeten System Calls sollten jeweils im **Manual** bzw. den entsprechenden Header-Files nachgelesen werden.
- Die Sprache C dient in erster Linie als Werkzeug zur Darstellung systemnaher Konzepte!

Inhaltsverzeichnis

Kapitel 1

Vorbemerkungen

- entwickelt 1972-73 von Dennis Ritchie bei den Bell Laboratories von AT&T
- 1978 erfolgten einige Erweiterungen von C (*enum, void, structure assignment, . . .*), die den sog. Kernighan/Ritchie-Standard (**K&R-Standard**) bilden
- Der ANSI-Standard beinhaltet eine Reihe von Erweiterungen und Aufräumarbeiten
- *C++* und *Objective C* sind modulare / objektorientierte Erweiterungen von C.
  C# (*C Sharp*) ist eine aktuelle Neuentwicklung (Mitte 2000) von Microsoft und vereint Konzepte von Visual Basic, Java und C++, ist aber stark mit der Windows-Plattform verbunden (*.NET*)
- UNIX ist zum größten Teil in C geschrieben, ein UNIX-Kern besteht nur aus 5-10% Assemblertext
- C ist eine maschinennahe Sprache. Array's sind Speicherflächen im Hauptspeicher, Array-Namen werden als Zeiger auf das erste Element aufgefaßt, Zugriff auf Elemente erfolgt via Zeigerarithmetik (Adress-Rechnung)
- C wurde mit UNIX verbreitet und ist damit eine der „portabelsten" Plattformen (*Achtung bei der Portabilität der C-Bibliotheken!*)
- In der Vorlesung und in den Übungen wird primär mit dem GNU-C(++)-Compiler gearbeitet, der auch unter Windows und DOS erhältlich ist.
- Literatur: jedes Buch, das sich mit C direkt beschäftigt (z.B. Kernighan/Ritchie: Programmieren in C, Hanser Verlag). Bücher über C++ sind aufgrund der Komplexität der objektorientierten Konzepte für die Vorlesung nicht empfehlenswert.

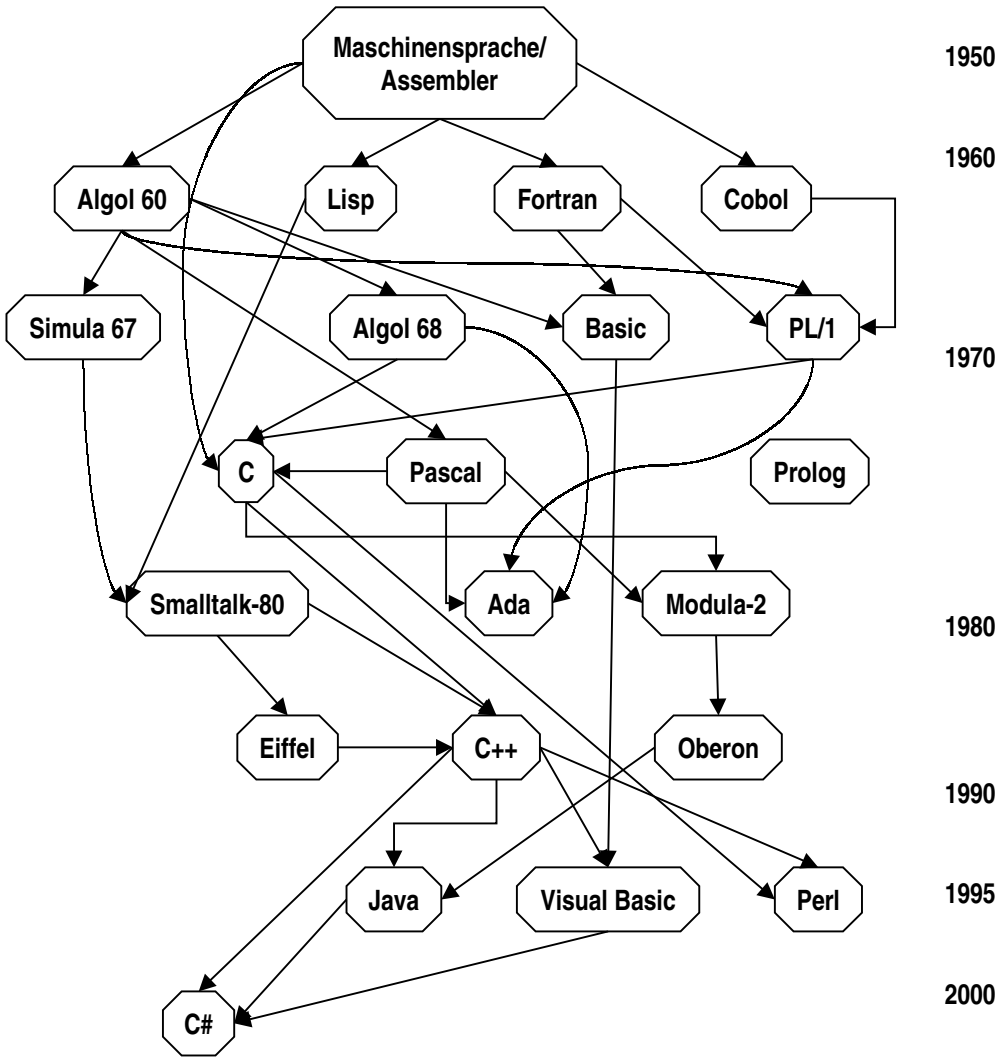

Abbildung 1.1: Übersicht Programmiersprachenentwicklung

Kapitel 2

Beispiel-Programme

2.1 Wie immer am Anfang

```
/* Ein erstes Beispiel: hallo.c */

main() {
/* puts = Ausgabe eines Strings nach stdout */
   puts("Tach Meister.");
}
```

Übersetzung und Ausführung:

```
byron$ gcc -Wall hallo.c  #-Wall: alle Compiler-Warnings einschalten
hallo.c:3: warning: return-type defaults to ‘int’
hallo.c: In function ‘main’:
hallo.c:5: warning: implicit declaration of function ‘puts’
hallo.c:6: warning: control reaches end of non-void function
byron$ a.out
Tach Meister.
byron$
```

Verbesserte Version:

```
/* hallo1.c */

#include <stdio.h>
/* in stdio.h ist u.a. die Funktion
 * puts vereinbart!
 */

int main() {
/* puts = Ausgabe eines Strings nach stdout */
   puts("Tach Meister.");
   /* Funktion main beenden und Exit-Status 0
    * an die Shell liefern! */
   return 0;
}
```

```
byron$ gcc -Wall -o hallo hallo1.c
byron$ hallo
Tach Meister.
byron$
```

Anmerkung: Der **Exit-Status** eines Programmes kann auf Shell-Ebene unmittelbar nach Beendigung des Programms durch das Kommando **echo $?** abgefragt werden. Üblicherweise wird bei Erfolg **0** zurückgegeben.

Anmerkung: C kennt keinen Datentyp **BOOLEAN**; der Integerwert **0** wird als **FALSE** interpretiert, jeder andere Wert als **TRUE**!

2.2 Fahrenheit nach Celsius

Mit einer **while**-Schleife (unter Verwendung von **const**, **unsigned int** und **float**):

```
#include <stdio.h>/* fahren.c   */

/* globale 'Konstanten'-Vereinbarungen: */
const unsigned int OG = 300, UG = 0;

int main()
{  int step = 20;
   float fahr, celsius;

   fahr = UG;
   while (fahr <= OG) {
      celsius = (fahr - 32.0) * (5.0/9.0);
      /* 5/9 waere ganzzahlige Division
       * und somit = 0, deshalb 5.0/9.0 */
      /* printf kann Zahlen formatiert ausgeben */
      printf("%4.0f Fahr. : %6.1f Celsius\n",
                    fahr, celsius);
      fahr = fahr + step;
   }
   return 0;
}
```

Mit einer **for**-Schleife

```
/* fahren1.c: Umwandlung von Fahrenheit in Celsius */
#include <stdio.h>

const unsigned int OG = 300, UG = 0, STEP = 20;

int main() {
   int fahr;

   printf("%4s     :       %7s\n\n", "FAHR","CELSIUS");
   for (fahr=UG; fahr <= OG; fahr=fahr+STEP)
      printf("%4d     :       %6.1f\n",
                  fahr, (5.0/9.0)*(fahr-32));
   return 0;
}
```

2.3 Euklid's Algorithmus

```
/*       euklid.c:      */
#include <stdio.h>

int main() {
   int x,y, x0,y0;

   printf("Geben Sie zwei pos. ganze Zahlen ein:");
   /* das Resultat von scanf ist die
    * Anzahl der eingelesenen Zahlen */
   if ( scanf("%d %d", &x, &y) != 2)
     return 1;

   x0 = x; y0 = y;
   while (x != y) {
      if (x > y)  /* kommt nur ein Statement, koennen */
         x = x-y; /* die { } weggelassen werden */
      else
         y = y-x;
   }
   printf("ggT von %d und %d ist %d\n",x0,y0,x);
   return 0;
}
```

An der Funktion **scanf()** sieht man, dass C nur Wertparameterübergabe kennt. Damit nun die in x und y eingelesenen Werte auch außerhalb von *scanf()* bekannt sind, muss von main an *scanf* ein *Zeiger* auf die Variablen (= Adresse im Hauptspeicher) übergeben werden. Damit ist der Zeigerwert zwar lokal zu *scanf()*, das Objekt, auf das gezeigt wird, ist aber in main bekannt, definiert und damit abrufbar. Mit dem Operator **&** wird ein **Zeiger** auf die folgende Variable „konstruiert". (scanf muss beim Zuweisen berücksichtigen, dass es sich bei den übergebenen Parametern um Zeigern auf Integervariablen handelt.)

Kapitel 3

Aufbau eines C-Programms

```
$ C_Program  = { Definition }.
$ Definition = FunctionDefinition | DataDefinition
$              | ExternDeclaration | TypeDefinition.
```

Syntax 3.1: Aufbau eines C-Programmes

- Ein C-Programm ist eine beliebige Folge von Vereinbarungen.
- Zu den Vereinbarungen gehören Funktionsdefinitionen, Typnamen-Vereinbarungen und Variablenvereinbarungen. Jede Funktionsdefinition hat einen Vereinbarungs- und Aktionsteil.
- Funktionen sind immer global, d.h. sie dürfen nicht geschachtelt werden (anders als in Modula-2 und Oberon!).
- Die Parameterübergabe ist immer **call-by-value**.
- Bei Prozeduren (Funktionen ohne Resultat) sollte **void** als **FunctionType** angegeben werden. Ist kein Ergebnistyp angegeben, so wird **implizit int** (*integer*) angenommen!

```
$ FunctionDefinition = FunctionType FunctionHeader
$                      CompoundStatement.
$ CompoundStatement  = "{" { Declaration }
$                          { statement } "}".
$ Statement          = [ Expression ] ";"
$                      | CompoundStatement | ... .
```

Syntax 3.2: Funktionsdefinition

Blockstruktur (anders als in Modula-2 oder Oberon):

Zu Beginn eines jeden **compound-Statement** dürfen Variablen-Vereinbarungen getroffen werden; damit können mehrere Anweisungen zusammengefaßt und Sichtbarkeits- / Lebensdauerbereiche definiert werden:

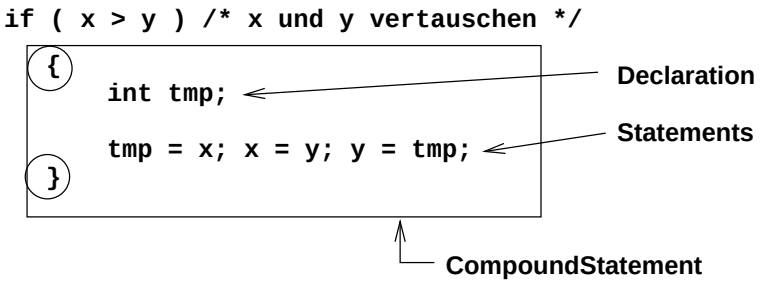

Abbildung 3.1: Compound Statement

- Die Gültigkeit von *tmp* erstreckt sich auf das *CompoundStatement*.
- Mit „**int tmp;**“ wird eine (lokale) Variablen mit Datentyp *int* deklariert. **int** ist ein Schlüsselwort und steht für *integer*.
- „**=**“ ist der „Zuweisungsoperator“ (Expression mit Nebeneffekt).
  „**tmp = x;**“ gilt als Ausdruck und hat einen Wert, nämlich den zugewiesenen Wert, also den von x.
  Eine Zuweisung der Form „**x = y = 1;**“ ist also möglich, die Bewertung des Ausdrucks erfolgt von rechts nach links:
  zuerst: „**y=1**“, d.h. y bekommt den Wert 1; der Wert des Ausdrucks ist der zugewiesene Wert (also 1);
  dieser Wert wird dann x zugewiesen; der Wert des Ausdrucks „**x=...**“ ist wiederum der zugewiesene Wert!

Kapitel 4

Etwas vorab zum C-Preprozessor

4.1 Motivation

Eine andere Technik zur Realisierung von Unterprogrammen ist der **offene Einbau** via **Makro** (Textersatz). Bei Aufruf eines normalenÜnterprogramms wird der zugehörige Anweisungsteil im compilierten Programm angesprungen, beim Aufruf durch einen Makro wird der Text des Unterprogramms an der Stelle einfach noch einmal vor dem Compilieren des Programms eingefgt. Der „Aufruf“ des Unterprogramms erfolgt durch Nennung des Makronamens (ggf. mit Parametern).
Vorteil: Abarbeitung des Aufrufs ist schneller (kein Anspringen, keine Stack-Verarbeitung)

Nachteil: Programmtext wird mit jeder Aufrufstelle größer

Das Programm, das diesen Textersatz durchführt, heißt **Makro-Prozessor** und ist bei Programm-Quelltexten dem eigentlichen Compiler vorgeschaltet; in diesem Zusammenhang spricht man dann vom **Preprocessor** („Präprozessor“).

4.2 cpp — der C-Preprocessor

Dem eigentlichen C-Compiler ist ein Makroprozessor vorgeschaltet, der automatisch mit dem Aufruf von **gcc** (und jedem anderen C-Compiler) aktiviert wird. Er kann auch direkt aufgerufen (gcc -E) werden (siehe *man*). Die von ihm erkannten Makros müssen als erstes Zeichen in der Zeile mit einem **#** beginnen, dem meist unmittelbar danach ein Makroname mit dem zugehörigen Ersatztext folgt. Der Ersatztext wird wiederholt auf evtl. Makroaufrufe untersucht.

```
#define OG 300 /* Ersatztext von OG ist 300 */
```

4.3 define-Makro

define ist ein Makroname, dessen Ersatztext „Nichts“ ist, der aber gleichzeitig eine Makrodefinition erledigt.

Beispiel:

```
/*      fahren2.c: mit Makro's statt const-Vereinbarung
        Umwandlung von Fahrenheit in Celsius
*/

#define OG 300
#define UG 0
#define STEP 20

int main() {
   float fahr, celsius;

   fahr = UG;
   while (fahr <= OG) {
       celsius = (fahr - 32.0) * (5.0/9.0);
       /* 5/9 waere ganzzahlige Division
          und somit 0, deshalb 5.0/9.0
        */
       printf("%4.0f Fahr. : %6.1f Celsius\n",fahr,celsius);
       fahr = fahr + STEP;
   }
   return 0;
}
```

cpp — Output:

```
# 1 "fahren2.c"




int main() {
   float fahr, celsius;

   fahr = 0 ;
   while (fahr <= 300 ) {
       celsius = (fahr - 32.0) * (5.0/9.0);



       printf("%4.0f Fahr. : %6.1f Celsius\n",fahr,celsius);
       fahr = fahr + 20;
   }
   return 0;
}
```

Will man die Kommentare bei der Ausgabe erhalten, dann kann man das durch Zuschalten der Option -C erreichen:

```
gcc -E -C fahren2.c
```

4.4 include

include ist ein Makro, dessen Ersatzttext der Inhalt der folgenden Datei ist. Damit werden i.a. Vereinbarungen oder andere Makro's „hereinkopiert". Diese Dateien heißen im Kontext mit C-Programmen **Header Files** und haben üblicherweise die Endung **.h**.

Es gibt eine ganze Reihe solcher Header Files im Katalog **/usr/include** — diese stellen die C-Bibliothek dar und entsprechen im Prinzip den *DEFINITION MODULE*'s von Modula-2 (resp. Oberon). Dies ist, wie später gezeigt wird, auch der Weg zur Modularisierung in C.

Da Makro's zu einem *inline*-Text führen, kann man damit auch Prozeduren realisieren, bei denen „kein Ansprung" erfolgt, die somit i.a. schneller sind. Dazu können Makro's auch parametrisiert werden (dazu und zu den weiteren Inhalten der *Header Files* später mehr).

Kapitel 5

Quellformat

5.1 Kommentare

Kommentare beginnen mit „/*“, enden mit „*/“, und können (im ANSI-Standard) nicht geschachtelt werden.

5.2 Namen

Namen bestehen aus Buchstaben und Ziffern, wobei das erste Zeichen ein Buchstabe sein muß. Zu den Buchstaben wird der *Underscore* (also „ _ “) gezählt.

5.3 Reservierte Worte

Die folgende Tabelle enthält die wichtigsten **Schlüsselworte** von C - einige wie **auto** oder **register** sind heute i.a. ohne Bedeutung.

`break`	`case`	`char`	`const`	`continue`
`default`	`do`	`double`	`else`	`enum`
`extern`	`float`	`for`	`goto`	`if`
`int`	`long`	`return`	`short`	`signed`
`sizeof`	`static`	`struct`	`switch`	`typedef`
`union`	`unsigned`	`void`	`while`	

Abbildung 5.1: C — Schlüsselworte

5.4 Operatoren — Übersicht

1	**primary**	`()  []  .  ->`	**left**
2	**unary**	`!  ~  -  ++  --  &  *` `( type )  sizeof`	**right**
3	**multiplicative**	`*  /  %`	**left**
4	**additive**	`+  -`	**left**
5	**shift**	`<<  >>`	**left**
6	**relational**	`<  <=  >  >=`	**left**
7	**equality**	`==  !=`	**left**
8	**bitwise AND**	`&`	**left**
9	**bitwise OR (excl.)**	`^`	**left**
10	**bitwise OR (incl.)**	`|`	**left**
11	**logical AND**	`&&`	**left**
12	**logical OR**	`||`	**left**
13	**conditional**	`?:`	**right**
14	**assignment**	`=  +=  -=  *=  /=  %=` `>>=  <<=  &=  ^=`	**right**
15	**comma**	`,`	**left**

Abbildung 5.2: C — Operatoren

Kapitel 6

Ein-/Ausgabe (stdin, stdout, stderr)

6.1 Ausgabe nach stdin - printf()

`nbytes = printf(formatstring, wert1, wert2, ...)`

- `nbytes:` printf liefert die Anzahl der ausgegeben Zeichen zurück.
- Ein `formatstring` ist eine in Doppelapostrophen eingeschlossene Zeichenfolge (konstanter String) oder Variable (vom Typ: Zeichenkette). Er besteht aus Text und eingestreuten Formatierungsangaben.
  Bsp. `printf(''Ich bin %d Jahre alt'', 3);`
- Formatierungsangaben beginnen mit einem **%** gefolgt von einem Zeichen zur Format-Modifikation und einem Zeichen zur Konvertierung.
- mgliche Werte für die Formatanweisungen

Zeichen für Konvertierung	Effekt
`d,i,o,u,x,X`	für Integer-Ausgabe; d und i für Dezimaldarstellung; u für unsigned; x,X für Hexadezimal, wobei x die Kleinbuchstaben a-f, X die Großbuchstaben A-F benutzt; unmittelbar davor kann ein h (für short int) oder l (für long int) stehen
`f`	Reelle Zahlen in Dezimalform ([-]mmm.nnnnnnn); Anzahl der Nachkommastellen durch Genauigkeitsangabe; default: 6
`e,E`	Reelle Zahlen in Exponentialform
`c`	gibt ein Zeichen aus; das zugehörige Datenargument wird als int übergeben, das letzte Byte wird ausgegeben
`s`	Ausgabe von Strings; das Argument muß ein Character-Vektor oder konstanter String mit jeweils abschließendem Null-Byte sein; Genauigkeitsangabe wird als maximal auszugebende Zeichenzahl interpretiert Bsp.: printf(“dieser String ist %s Zeichen lang“, “fuenfunddreissig“);
`%`	die Folge %% gibt ein % aus

Flags (opt.)	Bedeutung
-	linksbündig
+	auch pos. Vorzeichen wird ausgegeben
Blank	statt pos. Vorzeichen ein Blank

Minimale Feldgröße (optional):

Dezimalkonstante, die die minimale Anzahl auszugebender Stellen bestimmt; mit einem ***** wird das nächste Datenargument als minimale Feldgröße benutzt:

Beispiel:

```
thales$ cat feldgr.c
# include <stdio.h>

void main() {
   printf("%%5d\t%5d\n",25);
   printf("%%5.3d\t%5.3d\n",25);
   printf("%%.3d\t%.3d\n",25);
   printf("%%4.3f\t%4.3f\n",33.1);
   printf("%%4.3f\t%4.3f\n",33.123456789);
   printf("%%4.3e\t%4.3e\n",33.123456789);
   printf("%%.10s\t%.10s\n","Gib nur die\
   ersten 10 Zeichen aus.");
   printf("%%*d\t%*d\n",5,25);
}
thales$ gcc feldgr.c
thales$ a.out
%5d    25
%5.3d   025
%.3d 025
%4.3f 33.100
%4.3f 33.123
%4.3e 3.312e+01
%.15s Gib nur d
%*d    25
thales$ a.out | od -bc > feldgr.dump
thales$ cat feldgr.dump # etwas vereinfacht
045 065 144 011 040 040 040 062 065 012 045 065 056 063 144 011
  %   5   d  \t               2   5  \n   %   5   .   3   d  \t
040 040 060 062 065 012 045 056 063 144 011 060 062 065 012 045
          0   2   5  \n   %   .   3   d  \t   0   2   5  \n   %
064 056 063 146 011 063 063 056 061 060 060 012 045 064 056 063
  4   .   3   f  \t   3   3   .   1   0   0  \n   %   4   .   3
146 011 063 063 056 061 062 063 012 045 064 056 063 145 011 063
  f  \t   3   3   .   1   2   3  \n   %   4   .   3   e  \t   3
056 063 061 062 145 053 060 061 012 045 056 061 065 163 011 107
  .   3   1   2   e   +   0   1  \n   %   .   1   0   s  \t   G
151 142 040 156 165 162 040 144 151 012 045 052 144 011 040 040
  i   b       n   u   r       d   i  \n   %   *   d  \t
040 062 065 012
      2   5  \n
thales$
```

6.2 Ausgabe nach stderr - fprintf()

`nbytes=fprintf(stream, formatstring, wert1, wert2, ...)`

Im Prinzip wie *printf()* — zusätzlich muß als erstes Argument die Angabe der Ausgabeverbindung erfolgen (File Pointer). Für Diagnosemeldungen ist in **stdio.h** die Variable **stderr** definiert (manchmal auch als Makro)!

Beispiel:

```
hypatia$ cat diagnose.c

/* diagnose.c */

# include <stdio.h>

void main() {
  fprintf(stderr, "Fehlermeldungen wie Diagnosehinweise dahin\n");
  printf("Die ermittelten Ergebnisse dorthin\n");
}

hypatia$ gcc -Wall diagnose.c

hypatia$ a.out >res 2>err

hypatia$ cat res
Die ermittelten Ergebnisse dorthin
hypatia$ cat err
Fehlermeldungen wie Diagnosehinweise dahin
hypatia$
```

6.3 Eingabe von stdin - scanf()

`anzahl = scanf(formatstring, Adresse1, Adresse2, ...)`

- Format-String: bestehend aus Zeichen für die Konvertierung und weiteren Zeichen
- Konvertierung:
  % gefolgt von optionalen Zeichen zur Modifikation, gefolgt von Konvertierungszeichen
- Andere Zeichen (außer Konvertierungszeichen und Leeraum) müssen mit Zeichen im Eingabestrom übereinstimmen. Leerzeichen, Tab's (\t) und Newline (\n) veranlassen *scanf()* zum nächsten Nicht-Leerzeichen weiterzugehen.
- Beachte : C kennt nur Wertparameterübergabe. Aus diesem Grund muß speziell bei *scanf()* der **Adressoperator** vor den aktuellen Parameter gestellt werden (Ausnahme: Vektoren - dazu später mehr). Damit wird ein Zeiger auf die Variable übergeben und *scanf()* greift über diesen Zeiger auf das „Orginal“ durch!

Beispiel:

```
scanf("%d", &n);
```

Damit wird in die Integer-Variable n ein Wert von der *Standardeingabe* eingelesen.

scanf() hat einen ganzzahligen *return*-Wert, der die Anzahl der tatsächlich erfolgten Variablenzuweisungen wiedergibt.

Konvertierungs-Zeichen	Wirkung
d	Dezimal-Konstante
o	Oktal-Konstante
x,X	Hexadezimal-Konstante
f	Floating-Point
s	String; Zeichenfolge bis zum nächsten Leerraum (Blank, '\t', '\n'); Null-Byte ('\0') wird angefügt
c	Nächstes Zeichen; um das nächste Nicht-Leerraum-Zeichen zu lesen: %1c
%	Liest %-Zeichen ohne Zuweisung

- **Maximale Feldlänge:** Dezimal-Zahl
- **Flag: „*“** liest, unterdrückt aber Zuweisung

```
thales$ cat scan.c
/*** scan.c ***/

#include <stdio.h>

void main() {
  int i, digits, anzahl;
  float pi, x;
  char name[50];

  anzahl = scanf("Der Wert von pi auf %d Stellen ist %f",
         &digits, &pi);
  printf("%d Arg., pi: %f\n", anzahl, pi);
  anzahl = scanf("%2d %f %*d %2s", &i, &x, name);
  printf("%d Args, i: %d x: %f name: %s\n",
         anzahl, i, x, name);
}
thales$ gcc scan.c
thales$ a.out
Der Wert von pi auf 2 Stellen ist 2.17
2 Arg., pi: 2.170000
12345 0123 12a34
3 Args, i: 12 x: 345.000000 name: 12
thales$ a.out
Der WWWert von pi auf 2 Stellen ist 2.17
0 Arg., pi: 0.000000
0 Args, i: -268436676 x: 0.000000 name:
thales$ a.out
123 Der Wert von pi auf 2 Stellen ist 2.17
```

```
0 Arg., pi: 0.000000
2 Args, i: 12 x: 3.000000 name:
thales$
```

6.4 Weitere Ein-/Ausgabe-Funktionen

fgetc() – fgets() – getc() – getchar() – gets() – ungetc()

```
SYNOPSIS
       #include <stdio.h>

       int fgetc(FILE *stream);
       char *fgets(char *s, int size, FILE *stream);
       int getc(FILE *stream);
       int getchar(void);
       char *gets(char *s);
       int ungetc(int c, FILE *stream);

DESCRIPTION
       fgetc() reads the next character from stream and returns
       it as an unsigned char cast to an int, or EOF on end of
       file or error.

       getc() is equivalent to fgetc() except that it may be
       implemented as a macro which evaluates stream more than
       once.

       getchar() is equivalent to getc(stdin).

       gets() reads a line from stdin into the buffer pointed to
       by s until either a terminating newline or EOF, which it
       replaces with '\0'.  No check for buffer overrun is
       performed.

       fgets() reads in at most one less than size characters
       from stream and stores them into the buffer pointed to by
       s. Reading stops after an EOF or a newline. If a newline
       is read, it is stored into the buffer. A '\0' is stored
       after the last character in the buffer.

       ungetc() pushes c back to stream, cast to unsigned char,
       where it is available for subsequent read operations.
       Pushed-back characters will be returned in reverse
       order; only one pushback is guaranteed.

       Calls to the functions described here can be mixed with
       each other and with calls to other input functions from
       the stdio library for the same input stream.

RETURN VALUES
       fgetc(), getc() and getchar() return the character read as
       an unsigned char cast to an int or EOF on end of file or
```

```
       error.

       gets() and fgets() return s on success, and NULL on error
       or when end of file occurs while no characters have been
       read.

       ungetc() returns c on success, or EOF on error.

CONFORMING TO
       ANSI - C, POSIX.1

BUGS
       Because it is impossible to tell without knowing the data
       in advance how many characters gets() will read, and
       because gets() will continue to store characters past the
       end of the buffer, it is extremely dangerous to use. It
       has been used to break computer security. Use fgets()
       instead.

       It is not advisable to mix calls to input functions from
       the stdio library with low - level calls to read() for the
       file descriptor associated with the input stream; the
       results will be undefined and very probably not what you
       want.

SEE ALSO
       read(2),  write(2), fopen(3), fread(3), scanf(3), puts(3),
       fseek(3), ferror(3)
```

fputc() – putc() – putchar() – fputs()

```
SYNOPSIS
       #include <stdio.h>

       int fputc(int c, FILE *stream);
       int fputs(const char *s, FILE *stream);
       int putc((int c, FILE *stream);
       int putchar(int c);
       int puts((ccoonnsstt char **_s);

DESCRIPTION
       fputc() writes the character c, cast to an unsigned char,
       to stream.

       fputs() writes the string s to stream, without its trai-
ling '\0'.

       putc() is equivalent to fputc() except that it may be
       implemented as a macro which evaluates stream more than
       once.

       putchar(c) is equivalent to putc(c,stdout).

       puts() writes the string s and a trailing newline to stdout.
```

```
       Calls to the functions described here can be mixed with
       each other and with calls to other output functions from
       the stdio library for the same output stream.

RETURN VALUES
       fputc(), putc() and putchar() return the character written
       as an unsigned char cast to an int or EOF on error.

       puts() and fputs() return a non-negative number on suc-
cess,
       or EOF on error.

CONFORMING TO
       ANSI - C, POSIX.1

BUGS
       It is not advisable to mix calls to output functions from
       the stdio library with low-level calls to write() for
       the file descriptor associated with the same output
       stream; the results will be undefined and very probably
       not what you want.

SEE ALSO
       write(2), fopen(3), fwrite(3), scanf(3), gets(3),
       fseek(3), ferror(3)
```

Kapitel 7

Kontrollstrukturen

7.1 Übersicht

```
$ anweisung = prefix anweisung
$             | [ ausdruck ] ";"
$             | "break;"
$             | "continue;"
$             | "return" [ ausdruck ] ";"
$             | "goto" name ";"
$             | block
$             | "if" "(" ausdruck ")" anweisung
$             | "if" "(" ausdruck ")" anweisung
$               "else" anweisung
$             | "do" anweisung
$                    "while" "(" ausdruck ")" ";" .
$ prefix    = name ":"
$             | "while" "(" ausdruck ")"
$             | "for" "(" [ausdruck] ";"
$                         [ausdruck] ";" [ausdruck] ")"
$             | "switch" "(" ausdruck ")"
$             | "case" konstante ":"
$             | "default" ":" .
$ block     = "{" {lokal} {anweisung} "}" .
```

Syntax 7.1: Kontrollstrukturen

7.2 Abschluss von Anweisungen

- Semikolon „ ; “

Im Gegensatz zu anderen Sprachen wie z.B. Oberon – das Semikolon trennt dort Anweisungen voneinander – muss jede Anweisung durch ein Semikolon abgeschlossen werden! Bei zwei aufeinanderfolgenden Semikolons schliesst das zweite die **leere Anweisung** ab.

7.3 Bewertung eines Ausdrucks

Ist *ausdruck* ein bewertbarer Ausdruck (z.B. Zuweisung oder Funktionsaufruf), so wird dieser durch *ausdruck;* bewertet.

7.4 break-Anweisung

Dient zum vorzeitigen Verlassen einer Schleife oder zum „Verlassen“ eines Falles in der Mehrfachfallunterscheidung (**case**–Fall in der **switch**–Anweisung; **break** muss unbedingt angegeben werden, ansonsten wird auch der nächste case-Fall abgearbeitet)!

7.5 continue-Anweisung

Dient dazu, vorzeitig an den Schleifenanfang zu springen.

7.6 return-Anweisung

Wie in Modula-2 / Oberon: beendet Funktion und liefert Wert des danach stehenden Ausdrucks, falls vorhanden

7.7 if-Anweisung

Wahrheitswerte und Werte von Ausdrücken:

BOOL	int-Wert
TRUE	1 bzw. ungleich 0
FALSE	0

Beispiel für eine Fehlerquelle:

```
if (j=5) tu_etwas();

if (j==5) tu_etwas();
```

Die erste Version ist syntaktisch korrekt; *tu_etwas()* wird immer aufgerufen; die Bedingung **(j=5)** hat als Nebeneffekt die Wertzuweisung von 5 an die Variable j, der Wert der Bedingung ist der zugewiesene Wert, also 5, d.h. ungleich 0, also TRUE
subsectionif—else-Anweisung
Das **else** wird jeweils dem „nächsten“ **if** zugeordnet

7.8 switch-Anweisung (Mehrfachverzweigung)

```
/* switch.c */
#include <stdio.h>

int main() {
   int i;
   printf("Geben Sie eine Ganze Zahl: \n");
   while( scanf("%d", &i) > 0 ) {
      switch (i) {
      case 0:
              printf("0 eingegeben\n");
              break;
      case 1:
              printf("1 eingegeben\n");
              break;
      default:
              printf("weder 0 noch 1\n");
      }
      printf("Noch eine Zahl? \n");
   }
   return 0;
}
```

Zur Semantik und Verwendung:

- Der Ausdruck nach *switch* muß von integralem Typ sein (*char, short, int, long, enum*) — nicht *float, double, long double*

  nach K&R war nur *int* erlaubt
- Der Ausdruck nach **case** muß ein konstanter integraler Ausdruck sein
- Der *switch*-Ausdruck wird ausgewertet, das *case*-Label, das zum Ergebnis passt, wird angesprungen, der Kontrollfluss fährt beim zugehörigen Statement fort; d.h. insbesondere, daß auch die folgenden „Fälle“ abgearbeitet werden, falls dies nicht durch ein explizites **break** verhindert wird.
- Die *case*-Labels müssen verschieden sein
- Trifft keiner der Fälle zu, geht es bei **default** weiter (muß zwar nicht, sollte aber grundsätzlich der letzte Fall sein).
- Der *default*-Fall sollte immer vorhanden sein.
- Auch der *default*-Fall sollte mit **break** verlassen werden (spätere Änderungen!)

7.9 while-Anweisung

Beispiel: Zeichenweises Lesen von **stdio** und Zählen der Blanks

Die Bibliotheks-Funktion **getchar()** aus **stdio.h** liest ein Zeichen von *stdio* und liefert es als *int*-Wert. Das Ende des Eingabestroms ist über das Makro **EOF** (in *stdio.h*) als -1 definiert!

```
/*   punct.c */
# include <stdio.h>

int is_punc(char arg) {
/*   This function returns 1 if the argument is
 *   a punctuation character, otherwise zero
 */

   switch (arg) {
      case '.':
      case ',':
      case ':':
      case ';':
      case '!':  return 1;
      default:   return 0; break;
   }
}

int main() {
   char z;

   printf("Give one character:");
   if ( scanf("%c",&z) > 0) {
      if ( is_punc(z) )
         printf("punctuation character!\n");
      else
         printf("no punct. char.!\n");
      return 0;
   } else {
      printf("\nNichts eingegeben!\n");
      return 1;
   }
}
```

```
/*** getchar.c ***/

#include <stdio.h>

int main() {
   int ch, num_of_spaces = 0;

   while ( (ch = getchar()) != EOF)
      if (ch == ' ')
          num_of_spaces++;
   printf("Number of Blanks: %d\n", num_of_spaces);
   return 0;
}
```

7.10 do—while-Anweisung

```
/*** do_while.c ***/

#include <stdio.h>

int main() {
   int ch, num_of_spaces = 0;
   printf("Give one sentence:\n");

   do
      if ( (ch = getchar()) == ’ ’)
        num_of_spaces++;
   while ( ch != ’\n’);

   printf("Number of spaces: %d\n", num_of_spaces);
   return 0;
}
```

7.11 for-Anweisung

```
for(init-ausdruck; term-ausdruck; inkr-ausdruck)
                 anweisung;
```

Dies ist äquivalent zu

```
init-ausdruck; /* Initialisierung */
while (term-ausdruck)  /* Test */ {
   anweisung;
   inkr-ausdruck;  /* Inkrementierung */
}
```

Jeder der drei Ausdrücke kann leer sein; ein „leerer“ Test stellt eine stets erfüllte Bedingung (Endlos-Schleife) dar — mit **break** oder **return** zu verlassen.

Beispiel: Ziffernfolge einlesen und Zahlenwert ausgeben

```
/*
 *   for1.c
 */

#include <stdio.h>
#include <ctype.h> /* wegen Funktion isdigit */

int make_int() {
   int num = 0, d;

   for (d=getchar(); isdigit(d);d=getchar()) {
      num *=10; num += d - '0';
   }
   return num;
}

int main() {
   printf("Give a sequence of digits:\n");
   printf("The number is: %d\n", make_int());
   return 0;
}
```

```
/*** for2.c ***/
#include <stdio.h>
#include <ctype.h> /* wegen isdigit */
int make_int() {
   int num = 0, d;
   while ( isdigit( d = getchar() ) ) {
      num *=10; num += d - '0';
   }
   return num;
}
int main() {
   printf("Give a sequence of digits:\n");
   printf("The number is: %d\n", make_int());
   return 0;
}
```

```
/*** ignore.c ***/
#include <stdio.h>
#include <ctype.h>
void skip_spaces() {
   int c;
   for ( c=getchar(); isspace(c); c=getchar() ) ;
   ungetc(c,stdin);
}
int main() {
   int c;
   while ( (c = getchar()) != EOF)
      if ( c == ' ') {
         putchar(c); skip_spaces();
      } else
         putchar(c);
   return 0;
}
```

Kapitel 8

Operanden

8.1 Objekte und L–Werte

Allgemein betrachtet sind Objekte modifizierbare Speicherflächen mit einer internen Struktur (z.B. Array). L–Werte sind Ausdrücke, die ein *veränderbares* Objekt bezeichnen (z.B. Variablenname – keine Konstanten!): Da die Wertzuweisung in C ein Ausdruck ist, steht der Begriff **L**–Wert für die **linke** Seite in diesem Ausdruck, in dem als Nebeneffekt eben diese linke Seite verändert wird.

```
if (a = 7) printf("a = \%d\n", a);
```

Hier wird der Variablen a der Wert 7 zugewiesen (L-Value) und dann die Wertzuweisung als Ausdruck mit dem Wert 7 (ungleich 0, also TRUE) gewertet.

L–Werte können Operanden und Resultatwerte von unitären Operatoren (Auswahl eines Vektorelements oder einer Strukturkomponente) sein — diese können somit auch links vom „Zuweisungsoperator“ stehen. Dies gilt besonders für den **Dereferenzierungsoperator ***: ist p ein Zeigerwert, so ist ***p** das Objekt, auf das **p** zeigt.

8.2 Operanden im Einzelnen

```
$ operand = name | constant | string
$           | "(" expression ")"
$           | operand"("[argumentlist]")"
$           | operand"["expression"]" | operand"."name
$           | operand"->"name.
$ argumentlist = assignment {"," assignment}.
```

Syntax 8.1: Operanden

- **name**
  - ein Operand, meist L–Wert, wenn geeignet vereinbart
  - Typ folgt aus der Namensvereinbarung
  - ist er in einer Aufzählung eingeführt (Aufzähltyp), so ist es eine Konstante und somit kein L–Wert.

  - bezeichnet er einen Vektor, so gilt er als Adresse des ersten Elements (konstanter Zeigerwert, kein L–Wert)
  - bezeichnet er eine Strukturvariable, so ist er ein L–Wert
  - bezeichnet er eine Funktion (nicht beim Aufruf), so ist er eine Adresse (konstanter Zeigerwert)

- **constant**
  - ihr Typ ergibt sich aus der Definition, kein L–Wert;
  - Zeichen (Characters) sind vom Typ **int**

- **string**
  - repräsentiert einen konstanten Zeigerwert auf das erste Zeichen, kein L–Wert

- **argumentlist**
  - kann fehlen, runde Klammern aber dennoch notwendig;
  - Wertparameter
  - Reihenfolge der Bewertung **nicht** definiert

- **Vektorindizierung**
  - ganzzahlig, unterster Index ist **0**

Kapitel 9

Die Operatoren im Einzelnen

9.1 Unitäre Operatoren

```
$ unitaer = operand | operand"++"
$           | operand"--" | "*"unitaer
$           | "&"unitaer | "-"unitaer
$           | "!"unitaer | "~"unitaer
$           | "++"unitaer | "--"unitaer
$           | "("typangabe")" unitaer
$           | "sizeof" unitaer
$           | "sizeof""("typangabe")".
```

Syntax 9.1: Operatoren

- **Inkrement** $++$, **Dekrement** $--$: Argument muß L–Wert sein
- `(a)++ (a)--`
  liefern als Ergebnis den Wert **vor** Inkrementierung / Dekrementierung
- `++(a) --(a)`
  liefern als Ergebnis den Wert **nach** Inkrementierung / Dekrementierung
- ***** — **Derefenzierungsoperator**
- **&** — **Adressoperator**, Argument ist L–Wert
- **!** — logisches Komplement („Negation“); Ergebnis ist 1, falls Operand den Wert 0 hatte, 0 sonst
- **sizeof** — liefert die Zahl der Bytes, die ein Datentyp T (*sizeof(T)* oder ein Ausdruck x (*sizeof x* oder *sizeof(x)* belegt, bei einem Vektor die Größe des gesamten Vektors
- `~` — Bitweises Komplement (1-er Komplement)
- Beispiel zum bitweisen 1-er Komplement:

```
thales$ cat komplement.c
/*
 * komplement.c
```

```
 */

void main() {
   long int i;
   long int j;
   i=0;
   printf("i= %o (dez.: %d) und 1-Komplement: %o\n", i,i,~i);
   i=1;
   printf("i= %o (dez.: %d) und 1-Komplement: %o\n", i,i,~i);
   i = 2;
   printf("i= %o (dez.: %d) und 1-Komplement: %o\n", i,i,~i);
   i = 1024;
   printf("i= %o (dez.: %d) und 1-Komplement: %o\n", i,i,~i);

   j=1;
   for(i=1;i<31;i++)
    j = j*2;
   printf("j= %o (dez.: %d) und 1-Komplement: %o\n", j,j,~j);
}

thales$ gcc komplement.c
thales$ a.out
i= 0 (dez.: 0) und 1-Komplement: 37777777777
i= 1 (dez.: 1) und 1-Komplement: 37777777776
i= 2 (dez.: 2) und 1-Komplement: 37777777775
i= 2000 (dez.: 1024) und 1-Komplement: 37777775777
j= 10000000000 (dez.: 1073741824) und 1-Komplement: 27777777777
thales$
```

9.2 Binäre Operatoren

```
$ binaer = unitaer              | binaer "||" binaer
$          | binaer "&&" binaer | binaer "|" binaer
$          | binaer "^" binaer  | binaer "&" binaer
$          | binaer "==" binaer | binaer "!=" binaer
$          | binaer "<" binaer  | binaer "<=" binaer
$          | binaer ">" binaer  | binaer ">=" binaer
$          | binaer "<<" binaer | binaer ">>" binaer
$          | binaer "+" binaer  | binaer "-" binaer
$          | binaer "*" binaer  | binaer "/" binaer
$          | binaer "%" binaer.
```

Syntax 9.2: Binäre Operatoren

- && — logisches UND

  || — logisches ODER

  Operanden müssen mit Null vergleichbar sein; Resultat ist 0 oder 1; 0 entspricht **FALSE**; 1 (ungleich 0) entspricht **TRUE**

- & — **bitweises UND**,

  | — **bitweises inklusives ODER**,

  ^ — **bitweises exklusives ODER**

  Operanden müssen Integer sein, Ergebnis ist Integer

- **Vergleichsoperatoren**:

  - Resultat ist Integer,
  - 1, wenn Vergleich zutrifft,
  - 0, sonst

- % — Modulo-Operator

Beispiel:

```
byron$ cat bits.c
# include <stdio.h>

void main() {
   unsigned long int z1, z2;
   z1 = 5; z2 = 6;
   /*Bitmuster fuer z1: 00...000101
    *Bitmuster fuer z2: 00...000110
    */
```

```
    printf("z1=%ld (0101), z2=%ld(0110)\n", z1,z2);
    printf("z1 | z2: %ld\n", z1 | z2);
    printf("z1 & z2: %ld\n", z1 & z2);
    printf("z1 ^ z2: %ld\n", z1 ^ z2);
    printf("z1 << 2: %ld\n", z1 << 2);
    printf("z2 >> 1: %ld\n", z2 >> 1);
}
byron$ gcc -Wall bits.c
byron$ a.out
z1=5 (0101), z2=6(0110)
z1 | z2: 7
z1 & z2: 4
z1 ^ z2: 3
z1 << 2: 20
z2 >> 1: 3
byron$
```

9.3 Auswahl

```
$ auswahl = binaer | binaer "?" auswahl ":" auswahl.
```

Syntax 9.3: Auswahl-Operator

Die *auswahl* besteht aus einer Bedingung, der zwei Ausdrücke folgen. Die Bedingung wird bewertet: ist der Wert ungleich Null, so wird der erste Ausdruck als Ergebnis genommen, ist er gleich Null, so der zweite.

Beispiel:

```
byron$ cat auswahl.c
# include <stdio.h>

void main() {
  printf("%s\n", 2 > 3 ? "2 > 3" : "NOT 2 > 3");
}
byron$ gcc -Wall auswahl.c
byron$ a.out
NOT 2 > 3
byron$
```

Kapitel 10

Zuweisungen („Ausdrücke“)

```
$ zuweisung = auswahl
$             | unitaer "=" zuweisung
$             | unitaer "| =" zuweisung
$             | unitaer "^ =" zuweisung
$             | unitaer "& =" zuweisung
$             | unitaer "<< =" zuweisung
$             | unitaer ">> =" zuweisung
$             | unitaer "+ =" zuweisung
$             | unitaer "- =" zuweisung
$             | unitaer "* =" zuweisung
$             | unitaer "/ =" zuweisung
$             | unitaer "% =" zuweisung.
```

Syntax 10.1: Zuweisungen

a = *a* **op** *b* kann effizienter als *a* **op=** *b* angegeben werden, da *a* nur einmal bewertet werden muß.

Beispiele:

```
byron$ cat zuweisung.c
# include <stdio.h>
void main() {
   int a, b;
   a=7; b=15;

   printf("=========================================\n");
   printf("a= %2d; b= %2d\n",a,b);
   printf("-----------------------------------------\n");
   printf("a= %%2d : a = %2d\n", a);
   printf("a |= 0xF  -> a= %%2d: a=%2d\n", a |= 0xF);
   /* Reihenfolge der Parameterbewertung??? */
   printf("b= %%2d : b |= 0x3F -> b= %%2d: %d / %d\n",b, b |= 0x3F);
   a=5; b=17;
   printf("=========================================\n");
   printf("a= %2d; b= %2d\n",a,b);
   printf("-----------------------------------------\n");
   printf("a= %2d : ", a);
```

```
   printf("a &= 0x3  -> a= %2d\n", a &= 0x3);
   printf("b= %2d : b &= 0xF -> b= %2d\n",b, b |= 0xF);
}

byron$ gcc zuweisung.c
byron$ a.out
==========================================
a=  7; b= 15
------------------------------------------
a= %2d : a =  7
a |= 0xF  -> a= %2d: a=15
b= %2d : b |= 0x3F -> b= %2d: 63 / 63
==========================================
a=  5; b= 17
------------------------------------------
a=  5 : a &= 0x3  -> a=  1
b= 31 : b &= 0xF -> b= 31
byron$
```

Also Vorsicht!!!

Kapitel 11

Datentypen und Konstanten

11.1 Skalare Datentypen

Datentypen legen fest:

- den Speicherbedarf,
- die Interpretation der Bitbelegung sowie
- die erlaubten Operationen

Übersicht:

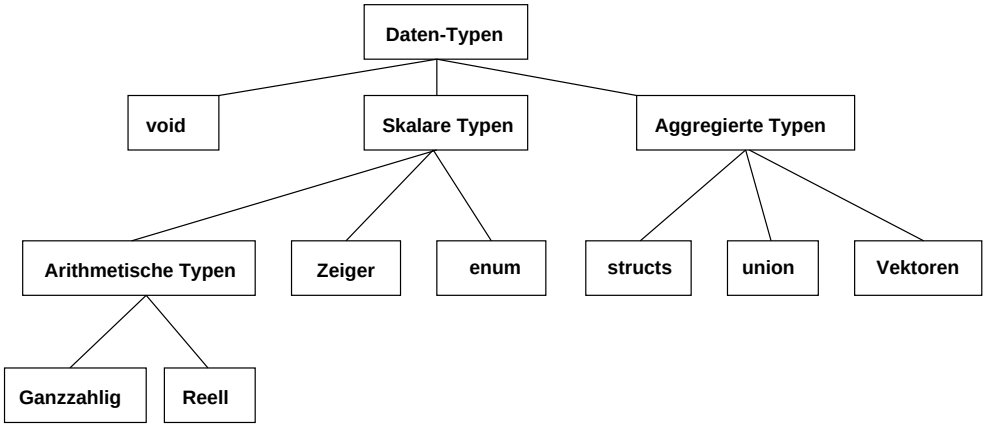

Abbildung 11.1: C — Datentypen

Skalare Typen:

- **Grundtypen** („Hauptworte“): **char**, **int**, **float**, **double**, **enum**
- **modifizierte Grundtypen** („Eigenschaftsworte“): **long**, **short**, **signed**, **unsigned**
- Der Datentyp **int** kann 2 oder 4 Byte umfassen, hardwareabhängig!

Konstante: Voranstellen von **const**

```
byron$ cat const.c
/* const.c */
# include <stdio.h>

void main() {
   const int i = 1;
   i++;
   printf("i = %d\n", i);
}
byron$ gcc -Wall const.c
const.c: In function ‘main’:
const.c:6: warning: increment of read-only variable ‘i’
byron$ a.out
i = 2
byron$
```

Wertebereiche (z.Zt.!):

Typ	Größe (Bytes)	Wertebereich
int	4 (oft auch nur 2)	$-2^{31} \ldots 2^{31} - 1$
short int	2	$-2^{15} \ldots 2^{15} - 1$
long int	4	$-2^{31} \ldots 2^{31} - 1$
unsigned short int	2	$0 \ldots 2^{16} - 1$
unsigned long int	4	$0 \ldots 2^{32} - 1$
signed char	1	$-2^{7} \ldots 2^{7} - 1$
unsigned char	1	$0 \ldots 2^{8} - 1$

11.2 Characters und Integers

Characters: 1 Byte Integers
Beispiel:

```
/* char_int.c */

#include <stdio.h>

void main() {
   char a,b; int i;
   a = 'A'; printf("a = %d\n", a);
   b = 65; printf("b = %c\n", b);
   i = 'A' + 1; printf("i = %c\n", i);
}
```

liefert:

```
byron$ gcc -Wall char_int.c
byron$ a.out
a = 65
b = A
i = B
byron$
```

11.3 „Mischen“ von Typen

Arithmetische Typen können in Ausdrücken mit wenigen Ausnahmen gemischt werden.

Explizite Konvertierung – der **(cast)**-Operator

Implizite (automatische) Konvertierung:

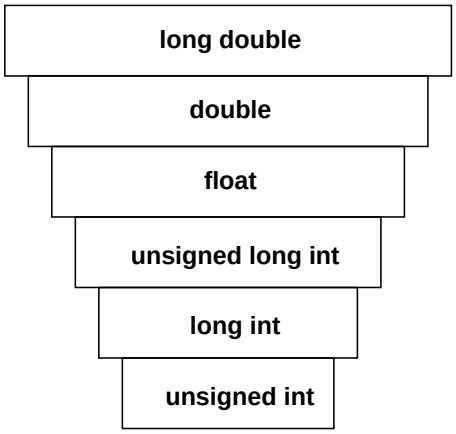

Abbildung 11.2: Typ-Hierarchie

- Zuweisungs-Konvertierung:

  Der Wert auf der rechten Seite wird konvertiert in den Typ auf der linken.

- Ganzzahlige Erweiterung:
  - *char* und *short int* in einem Ausdruck werden in *int* konvertiert.
  - *unsigned char* und *unsigned int* werden in *int* konvertiert (falls nicht zu groß, sonst *unsigned int*).
- In arithmetischen Ausdrücken erfolgt die Konvertierung so, dass die Konvertierungsregeln des Operators beachtet werden.
  - Enthält ein Operandenpaar eine *long double*, so wird der andere ebenfalls in *long double* konvertiert; ansonsten:
  - ist einer der beiden Operanden *double*, so wird der andere ebenfalls in *double* konvertiert; ansonsten
  - entsprechend obiger Hierarchie.

```
/* char_int2.c */
#include <stdio.h>
void main() {
   char c = 5; short j = 6; int k;

   k = c + j + 8; /* c und j werden zu
                   * int konvertiert
                   */
   printf("k = %d\n", k);

   { unsigned short l = 71124;
     printf("l = %d\n", l);
   }

   c = 882;
   printf("c = %c\n", c);
   c = 114;
   printf("c = %c\n", c);
}
```

liefert

```
byron$ gcc -Wall char_int2.c
char_int2.c: In function ‘main’:
char_int2.c:16: warning: large integer
                implicitly truncated to
                unsigned type
char_int2.c:20: warning: overflow in
                implicit constant conversion
byron$ a.out
k = 19
l = 5588
c = r
c = r
byron$
```

Anmerkungen:

- Die Binärdarstellung von 882 ist 00000011**01110010**
  Zugewiesen wird das „hintere“ Byte, also 114!
- Bei der Zuweisung von 71124 an j wird der Rest von $71124/(65535+1)$ genommen, also 5588!

11.4 Datentyp “char”

- Einzelne Zeichen (Konstanten) werden in einfache Hochkommas eingeschlossen
- Ersatzdarstellungen:

\b	BS	backspace
\f	FF	formfeed
\n	LF	newline, Zeilentrenner
\r	CR	carriage return, „Wagenrücklauf“
\t	HT	Horizontaler Tab
\\	\	„Fluchtsymbol“
\’	’	einfaches Hochkomma
\0	NUL	Null-Byte
\ddd		Bit-Muster (oktal)

11.5 Gleitkommawerte (float, double)

Beispiel (zur Sensibilisierung bzgl. Genauigkeit von Gleitkomma-Rechnungen):

```
/* genau.c : Auswertung von 9*x*x*x*x-y*y*y*y+2*y*y */
/* RICHTIGES ERGEBNIS = +1 */
# include <stdio.h>

void main() {
   int x  =  10864, y  =  18817, z;
   float x1 = x, y1 = y, z1;
   double x2, y2, z2, z2a, x3, y3, z3, u3, v3, w3;

   x3 = x2 = x1; y3 = y2 = y1;
   /************* als int: ************************/
   z  =  9*x*x*x*x-y*y*y*y+2*y*y ;
   printf("ganzzahlig: %d\n",z);
   /************ als float: ************************/
   z1 =  9*x1*x1*x1*x1-y1*y*y*y+2*y*y ;
   printf("als float: %e\n",z1);
   /****** als double, mit anderer Auswertung ********/
   z2  =  9*x2*x2*x2*x2 + 2*y2*y2;
   z2a  =  z2 - y2*y2*y2*y2;
   printf("als double, Formel umgestellt: %.12f\n",z2a);
   /************* double, schrittweise ***************/
   u3  =  9*x3*x3*x3*x3;
   v3  =  2*y3*y3;
   w3  =  u3+v3;
   z3  =  y3*y3*y3*y3;
   printf("u3  =  %g\n", u3); printf("v3  =  %g\n", v3);
   printf("w3  =  %g\n", w3); printf("z3  =  %g\n", z3);
   z3  =  w3 -z3;
   printf("und jetzt: %g\n", z3);
}
```

liefert

```
byron$ gcc genau.c
byron$ a.out
ganzzahlig: 1
als float: 1.000000e+00
als double, Formel umgestellt: -1.000000000000
u3 = 1.25372e+17
v3 = 7.08159e+08
w3 = 1.25372e+17
z3 = 1.25372e+17
und jetzt: 0
byron$
```

Noch ein Beispiel:

```
/* genau1.c */

#include <stdio.h>

void main() {
        double x,y;

        x = 2.0;
        y = (2.0/0.77777777777777);
        y *=     0.77777777777777;

        if (x==y)
                puts("gleich");
        else
                puts("ungleich");
        if (abs(x-y) < 1.0E-9)
                puts("gleich");
        else
                puts("ungleich");

}
```

liefert

```
hypatia$ gcc -Wall genau1.c
hypatia$ a.out
ungleich
gleich
hypatia$
```

11.6 Aufzählungen — enum

```
$ enumtype = "enum" [ name ] { enumerator {"," enumerator}}
$            | "enum" name .
$ enumerator = name [ "=" konstante ] .
```

Syntax 11.1: Aufzähltyp

- Steht zwischen **enum** und der Aufzählung ein *name*, so kann anschließend damit die Aufzählung bezeichnet werden.
- Der Wert der Konstanten beginnt von links nach rechts folgend mit 0
- Wird beim *enumerator* mit „ = “ eine Konstante angegeben, so wird obige Angabe (0, 1, . . .) überschrieben
- Verwendet man einen Aufzählungswert als Index in einem Vektor, so muß dieser explizit (*cast*) in *int* gewandelt werden.

```
/* enum.c */
#include <stdio.h>

void main() {
   /* Variable: */
   enum { rot, gelb, gruen } ampel;
   /* Typname: */
   enum farbe { blau, lila, pink };

   enum farbe wand;

   ampel = rot;
   if (ampel == rot )
      printf("Die Ampel ist rot!\n");

   wand = pink;
   if (wand == pink)
      printf("Die Wand ist pink-farben!\n");
}
```

11.7 Vektoren (Array's)

```
$ vector = type_spec vec_name "["vec_size"]"
$            { "["vec_size"]" } [ initializer ] .
```

Syntax 11.2: Vektoren

- Der **Vektroname** ist ein **konstanter Zeiger** auf das erste Element und wird so auch an Funktionen übergeben!
- Der Typ ist also ein Zeiger auf den Elementtyp, unabhängig von der Dimensionierung
- Die Indizierung beginnt bei **0** und geht bis **vec_size - 1**
- Problem: Indizierungsfehler werden u.U. erst zur Laufzeit erkannt (*segmentation violation*)!!!

Beispiel: Summe über Vektorelemente

```
/* vektor1.c */
#include <stdio.h>

const int Length = 10;

void main() {
   int vektor[Length]; int i, sum;

   for (i=0; i < Length; i++)
      vektor[i] = i;
   sum = 0;
   for (i=Length - 1; i >= 0; i--)
      sum += vektor[i];
   printf("Summe: %d\n", sum);

   /* Indizierungsfehler: */
   sum = 0;
   for (i=0; i <= Length+2; i++)
      sum += vektor[i];
   printf("Summe (falsch): %d\n", sum);
}
```

liefert

```
byron$ gcc -Wall vektor1.c
byron$ a.out
Summe: 45
Summe (falsch): 1073766905
byron$
```

Noch ein Beispiel:
liefert

```
/* vektor2.c */
#include <stdio.h>


void main() {
   int i, sum, length;
   int vektor[] = { 1, 2, 3, 4, 5 };
   length = sizeof(vektor) / sizeof(vektor[0]);
   printf("Length = %d\n", length);

   sum = 0;
   for (i=length-1; i >= 0; i--)
         sum += vektor[i];
   printf("Summe: %d\n", sum);

   /* etwas Bloedsinn: */
   vektor[5] = 6;

   sum = 0;
   for (i=length-1; i >= 0; i--)
          sum += vektor[i];
   printf("Summe: %d\n", sum);
}


byron$ gcc -Wall vektor2.c
byron$ a.out
Length = 5
Summe: 15
Summe: 21
byron$
```

Also aufgepaßt!!!

11.8 Zeiger

```
$ zeigervariable = el_type "*" var_name .
```

Syntax 11.3: Zeiger

Durchgriff auf referenziertes Objekt: durch Voranstellen des ***** Operators vor die Zeigervariable!

- Zeigerwerte können als *integer* interpretiert werden (Achtung: systemabhängig)
- Null-Zeiger: **0** oder manchmal auch **`(void*)0`**bzw. als Makro: **NULL** (in **stdlib.h**)
- Der Zeiger ist an den Typ gebunden, auf den gezeigt wird; kann aber explizit auf andere Typen gewandelt werden (**cast**)
- Der Wert einer Zeigervarablen ist die Adresse einer Speicherfläche, deren Inhalt von dem Typ sein muß (sollte), auf den der Zeiger zeigt.
- Entstehung des Wertes einer Speichervariablen:
  Zuweisung der Adresse einer bestehenden Speicherfläche vom entsprechenden Typ

  Dynamische Speicher-Allokation (siehe Funktionen aus **stdlib.h**: **calloc**, **malloc**, **realloc**)

Beispiel:

```
/* zeiger.c */
# include <stdio.h>
/* wegen calloc */
# include <stdlib.h>
void main() {
   int i = 10; int * p;
   int v[] = { 1,2,3,4,5 };
   p = &i; printf("*p = %d\n", *p);
   p = v; printf("*p = %d\n", *p);
   p = &v[0]; printf("*p = %d\n", *p);
   p = (int *) calloc(1,sizeof(int));
   if (p) {
      *p = 100; printf("*p = %d\n", *p);
   }
}
```

liefert

```
byron$ gcc -Wall zeiger.c
byron$ a.out
*p = 10
*p = 1
*p = 1
*p = 100
byron$
```

Zeiger-Arithmetik:

- Addition
  Sei ptr eine Zeigervariable: der Wert von $ptr + 3$ zeigt auf ein Objekt, das 3 „Plätze“ hinter $*ptr$ liegt; d.h., es wird um $3 * sizeof(*ptr)$ weitergegangen.
- Subtraktion
  nur Zeiger vom selben Typ
  liefert *int* Wert (Anzahl der Objekte dazwischen)

Beispiele:

```
long *p1, *p2;
int j;
char *p_ch;

p2 = p1 + 4;    /*zulaessig; sinnvoll?*/
j  = p2 - p1;   /*zulaessig; j == 4*/
j  = p1 - p2;   /*zulaessig; j == -4*/
p1 = p2 - 2;    /*zulaessig, gleicher Typ; sinnvoll?*/
p_ch = p1 - 1;  /*NICHT sinnvoll, versch. Typ*/
j  = p1 - p_ch; /*NICHT sinnvoll, versch. Typ*/
```

11.9 Vektoren als Parameter

Der Name eines Vektors wird als Zeiger auf das erste Element interpretiert. Übergibt man also den Namen eines Vektors als Parameter, so wird der Zeigerwert via Wertparameter übergeben; greift man über die lokale Kopie des Zeigerwerts auf den Vektor durch, ist man beim Orginal-Vektor.

```
/* vekt_zeiger.c */

#include <stdio.h>

void init(int ar[], int length) {
   int i;
   for(i=0; i < length; i++)
     ar[i] = i;
}

int summe1(int ar[], int length) {
   int i, sum = 0;
   for (i=0; i < length; i++)
      sum += ar[i];
   return sum;
}

int summe2(int * ar, int length) {
   int i, sum = 0;
   for (i=0; i < length; i++,ar++)
      sum += *ar;
   return sum;
}

void main() {
   const int vekt_len = 10;
   int vektor[vekt_len];

   init(vektor, vekt_len);

   printf("Summe: %d\n", summe1(vektor, vekt_len));
   printf("Summe: %d\n", summe2(vektor, vekt_len));
}
```

11.10 Zeichenketten — Strings

- String-Konstante: eine in Doppelapostroph eingeschlossene Zeichenfolge mit abschließendem Null-Byte (\0), wird i.a. vom Compiler angefügt
- String-Konstante werden i.a. im Read-Only-Speicherbereich abgelegt!!!

Beispiel:

```
/* strings.c */
#include <stdio.h>

char * str = "ein Text";

void main() {
    char array[10];
    char string[10] = "Hallo!";
    char *ptr1 = "10 char's";
    char *ptr2;

/*  array = "not OK";*/  /* nicht zulaessig, array ist
                          * konstante Adresse
                          */

     array[5]='A';       /* zulaessig */

/*   ptr1[5]='B'; */     /* i.a. nicht zulaessig, da die
                          * Speicherflaeche, auf die ptr1
                          * hier zeigt, READ ONLY ist!
                          */

      ptr1="ok";
      printf("ptr1: %s\n", ptr1);

      ptr2 = str;
      printf("ptr2: %s\n", ptr2);

      string[0]='X';
      printf("string=%s\n", string);
}
```

liefert

```
byron$ gcc -Wall strings.h
byron$ a.out
ptr1: ok
ptr2: ein Text
string=Xallo!
byron$
```

Einige Beispiele: (Die Funktionen **strcpy()** und **strcmp()** sind in **string.h** bereits definiert!)

```
void strcpy1(char s[], char t[]) {
/*t nach s kopieren */
        int i;
        i = 0;
        while ( (s[i] = t[i]) != '\0')
        {
                printf(" i: %d\n", i);
                i++;
        }
}


void strcpy2(char *s, char *t) {
        while ( (*s = *t) != '\0' )
        {
                s++;
                t++;
        }
}


void strcpy3( char *s, char *t ) {
        while( (*s++ = *t++) != '\0')
                ;
}


/* Noch kuerzer moeglich, da bei Zuweisung
 * des Null-Bytes der Wert des Ausdrucks 0 ist
 * und damit die Bedingung "FALSE"
*/
strcpy4( char *s, char *t ) {
        while (*s++ = *t++)
                ;
}


int strcmp1(char s[], char t[]) {
/*<0, wenn s<t, 0, wenn s=t, >0, wenn s>t */
        int i = 0;
        while ( s[i] == t[i] )
                if ( s[i++] == '\0')
                        return (0);
        return (s[i]-t[i]);
}


/* Zeigerfassung */
int strcmp2(char *s, char *t) {
        for ( ; *s == *t; s++, t++)
                if (*s == '\0')
                        return (0);
        return (*s - *t);
}
```

Kapitel 12

Mehrdimensionale Felder

12.1 Definition

Beispiel:

```
int matrix[2][3];
```

matrix ist ein 2-elementiger Vektor, dessen Elemente 3-elementige Vektoren sind, deren Elemente integer-Zahlen sind.

Zugriff auf ein „Matrix“-Element:

```
matrix[1][0]
```

Initialisierung:

```
int matrix[2][3] = { {0,1,2}, {3,4,5} }
```

Element	Adresse	Inhalt
matrix[0][0]	1000	0
matrix[0][1]	1004	1
matrix[0][2]	1008	2
matrix[1][0]	100C	3
matrix[1][1]	1010	4
matrix[1][2]	1014	5

Der Zugriff `matrix[1][2]` wird interpretiert als `*(matrix[1]+2)`, dies wiederum als `*(*(matrix + 1) + 2)`.

12.2 Mehrdimensionale Felder als Parameter

```
f1() {
int ar[5][6][7];
...
f2(ar);
...
}

f2( int erh_arg[][6][7]) {
...
}
```

`erh_arg` wird interpretiert als `int (*erh_arg) [6][7]`; `erh_arg` ist also ein Zeiger auf einen 6-dimensionalen Vektor, dessen Elemente 7-dimensionale Vektoren sind, deren Elemente wiederum *integer* sind.

Das Ganze mit Zeigern:

```
f1() {
int ar[4][5][6];
...
f2(ar,4,5,6);
...
}

f2( int *** erh_arg, int dim1, int dim2, int dim3) {
...
}
```

Zugriff auf `ar[x][y][z]`:
`* ( (int *) erh_arg + x*dim3*dim2 + y*dim2 + z)`
Der **cast** `(int *)` ist notwendig, da `erh_arg` Zeiger auf Zeiger ist und die Skalierung über Zeigergröße falsch wäre.

Kapitel 13

Dynamische Speicherverwaltung

- `void * calloc(unsigned nelem, unsigned elsize)`

  liefert einen Zeiger auf mit Null initialisierte Speicherfläche für *nelem* Objekte der angegebenen Größe. Ist nicht genügend Speicher verfügbar, wird der Null-Zeiger zurückgegeben. Mit einer entsprechenden *cast*-Operation kann der Zeiger auf den entsprechenden Typ gewandelt werden:

```
void * calloc();
int * i_p;

i_p = (int *) calloc(n, sizeof(int));
```

  Mehr dazu über **man calloc**

- `void * malloc(unsigned size)`

  liefert einen Zeiger auf eine Speicherfläche mit wenigstens *size* Bytes, nicht initialisiert. Achtung: `calloc` ist `malloc` vorzuziehen (Speicherinitialisierung und Implementierungsfehler in manchen Compilern!)

- `void * realloc(char *ptr, unsigned size)`

  ändert die Größe der bislang über *ptr* definierten Speicherfläche auf die angegebene Byte-Zahl und liefert einen Zeiger auf eine (möglicherweise „bewegte“) Speicherfläche zurück; der Inhalt bleibt soweit unverändert.

- `void free(char *ptr)`

  gibt eine zuvor durch *calloc(), malloc(), realloc()* besorgte Speicherfläche frei. In vielen Büchereien gibt es daneben die Funktion

  ```
  int cfree(char *ptr);
  ```

  Meist ist *free()* vom Ergebnistyp *int* und signalisiert mit 1 Erfolg und mit 0 Mißerfolg.

Beispiel: Namen von **stdin** in Tabelle einlesen und wieder ausgeben

```
/* table.c */

# include <stdio.h>
# include <stdlib.h>
# include <string.h>

#define TABLE 30
#define NAME 20

int main()
{    char *table[TABLE];    /* Platz fuer die Pointer */
     char name[NAME];       /* Eingabe vom Terminal */
     int i, jogg, len;

     for (i=0; i < TABLE; i++) {
        if (fgets(name,NAME,stdin)!=NULL) {
          len = strlen(name);       /* Laenge der Eingabe*/
          if (name[len-1] == '\n')  /* Newline loeschen  */
             name[len-1] = '\0';
          else                      /* line was too long */
             while(getchar()!='\n')
                ;                   /* skip to line end  */
                                    /* in Tabelle kopieren*/
          table[i] = (char *) calloc(NAME, len+1);
          strcpy(table[i], name);
        }
        else                        /* Eingabeende */
            break;
     }
     for (jogg=0; jogg <i; jogg++)  /* ausgeben&freigeben*/
          puts(table[jogg]), free(table[jogg]);

     return 0;
}
```

Kapitel 14

typedef

- Vereinbarung neuer Datentyp-Namen

Beispiel:

```
typedef int LENGTH;
```

Damit ist `LENGTH` synonym zu `int`.
Also kann statt `int i,j;` auch `LENGTH i,j;` verwendet werden.
Die Vereinbarung ist wie bei Variablenvereinbarungen: der neue Typ-Name steht an der Stelle des Variablennamens.

Mit *typedef* wird **kein** neuer Datentyp (wie in Modula-2 oder Oberon mit TYPE) konstruiert. Es wird nur ein Name für einen anderweitig auch existierenden Datentyp eingeführt.

typedef ist nützlich gegen Portabilitätsprobleme: führt man Datentypen mittels *typedef* ein, so muß ggf. nur die *typedef*-Vereinbarung geändert werden.

Mittels *typedef* kann ein Programm auch besser dokumentiert werden (sprechende Typ-Namen)
typedef verhält sich ähnlich wie **define** — nur wird *typedef* vom Compiler verarbeitet, während *define* vom Preprozessor verarbeitet wird. So ist

```
#define USHORT unsigned int
```

äquivalent mit

```
typedef unsigned int USHORT;
```

Aber:

```
#define PT_TO_INT int *
PT_TO_INT p1, p2;
```

wird expandiert zu `int * p1, p2;` damit ist nur `p1` ein `int`-Pointer, `p2` ist ein `int`-Objekt!
Mit

```
typedef int * PT_TO_INT;
PT_TO_INT p1,p2;
```

werden zwei `int`-Pointer eingeführt.

Kapitel 15

Strukturen (Records)

15.1 Vereinbarungen

Beispiel:

```
struct pers_dat {
   char *p_name, *p_nr ;
   short geb_monat, geb_tag, geb_jahr;
};

/* Variable pers: */
struct pers_dat pers;
```

Die Variable `pers` steht für die gesamte Struktur (Speicherfläche). `pers_dat` ist ein Name für die Struktur (keine Typ-Vereinbarung).

Struktur ohne Namen:

```
struct {
    char *p_name, *p_nr;
    short geb_monat, geb_tag, geb_jahr;
} person;
```

Mit Typvereinbarung:

```
/* Typ-Name fuer Struktur ist Pers_dat: */
typedef struct {
   char *p_name, *p_nr;
   short geb_monat, geb_tag, geb_jahr;
} Pers_dat;

/* Variable person vom Type pers_dat */
Pers_dat person;
```

15.2 Initialisieren bei Definition

```
Pers_dat person = {"Otto Huber","SAI",3,5,1950};
```

15.3 Zugriff auf Komponenten

```
if (person.geb_tag == 28) ...
```

15.4 Zeiger auf Strukturen

```
Pers-dat *p_to_person;
...

if( (p_to_pers->geb_tag >28) || ((*p_to_pers).geb_monat == 2 )){
 ...
}
```

15.5 Geschachtelte Strukturen

```
typedef struct {
       char * name, * vname;
       struct {
              short tag, monat, jahr;
       } geb_datum;
} Person;

Person p1;
 ...

p1.geb_datum.tag = 21;
```

oder so:

```
typedef struct {
       short tag, monat, jahr;
} Datum;

typedef struct {
       char * name, * vname;
       Datum geb_datum;
} Person;
```

15.6 Rekursive Strukturen

```
struct s {
       int a,b;
       float c;
       struct s * p_to_s;
};     /* Zeiger als Vorwaertsverweis ist ok! */

struct s1 {
       int a;
```

```
       struct s2 * b;
}

struct s2 {
       int v;
       struct s1 * w;
}
```

Eine der wenigen Ausnahmen von der Regel „declare before use“ !!!

ABER:

```
typedef struct {
    int a;
    fehler * p; /*Fehler, da fehler noch nicht def.*/
} fehler;
```

15.7 Strukturen als Funktionsargumente

```
/* struct1.c */

#include <stdio.h>

typedef struct {
        short tag, monat, jahr;
} Datum;

typedef struct {
        char name[21], vname[11];
        Datum geb_datum;
} Person;

typedef struct {
        Datum d;
        char ereignis[31];
} Ereignis;

Ereignis fest = {
                  {1,1,2000}, {"Neujahr"}
};

void ausgabe(Ereignis arg) {
        printf("Am %d.%d.%d ist ein besonderes %s\n", \
arg.d.tag, arg.d.monat, arg.d.jahr,
                arg.ereignis);
}

void main() {
        ausgabe(fest);
}
```

Mit Zeigern:

```
/* struct2.c */

#include <stdio.h>

typedef struct {
        short tag, monat, jahr;
} Datum;

typedef struct {
        char name[21], vname[11];
        Datum geb_datum;
} Person;

typedef struct {
        Datum d;
        char ereignis[31];
} Ereignis;

Ereignis fest = {
                  {1,1,2000}, {"Neujahr"}
};

void ausgabe(Ereignis * arg) {
        printf("Am %d.%d.%d ist ein besonderes %s\n", \
arg->d.tag, arg->d.monat, arg->d.jahr,
                arg->ereignis);
}

void main() {
        ausgabe(&fest);
}
```

15.8 Strukturen als Ergebnis von Funktionen

```
/* struct3.c */
#include <stdio.h>

typedef struct {
        short tag, monat, jahr;
} Datum;

typedef struct {
        Datum d;
        char ereignis[31];
} Ereignis;

Ereignis bestimme() {
   Ereignis er = {
            {1,1,2000},
            {"Neujahr"}
   };
   return er;
}

void ausgabe(Ereignis arg) {
   printf("Am %d.%d.%d ist / war ein spezielles %s\n",
      arg.d.tag, arg.d.monat, arg.d.jahr, arg.ereignis);
}
void main() {
   Ereignis fest = bestimme();
   ausgabe(fest);
}
```

Achtung bei der Rückgabe von Zeiger auf Strukturen:

```
/* struct4.c */
#include <stdio.h>

typedef struct {
        short tag, monat, jahr;
} Datum;

typedef struct {
        Datum d;
        char ereignis[31];
} Ereignis;

Ereignis * bestimme() {
   Ereignis er = {
            {1,1,2000},
            {"Neujahr"}
   };
   return & er;
}

void ausgabe(Ereignis arg) {
   printf("Am %d.%d.%d ist / war ein spezielles %s\n",
      arg.d.tag, arg.d.monat, arg.d.jahr, arg.ereignis);
}
void main() {
   Ereignis * fest = bestimme();
   ausgabe(* fest);
}
```

liefert:

```
hypatia$ gcc -Wall struct4.c
struct4.c: In function ‘bestimme’:
struct4.c:18: warning: function returns address of local variable
hypatia$ a.out
Am 1.1.2000 ist / war ein spezielles Ne
hypatia$
```

Erklärung: das Ereignis-Objekt `er` ist nur zur Laufzeit der Funktion `bestimme` definiert. Nach Beendigung der Funktion wird der Speicherplatz von `er` anderweitig verwendet und damit neu belegt.

15.9 Ein kleines Listen-Programm

```
/* list.h : Vereinbarungen und includes */
#include <stdio.h>
#include <stdlib.h>

typedef int BOOL;
#define FALSE 0
#define TRUE 1

typedef struct node *ListPtr;
typedef struct node { /*Listenelement */
   int count;
   ListPtr next;
} listNode;


ListPtr buildList( int );

void printListAsc(ListPtr);

void printListDesc(ListPtr);
```

Anm.: Dieses Header-File muss als

```
# include "list.h"
```

herangezogen werden. Die Apostrophen sind hier zu beachten (mehr dazu später)!

```
/* ein kleines Listenpogramm */
# include "list.h"

ListPtr buildList(int n) {
   int i = 1; ListPtr firstEl, lastEl;

   firstEl = (ListPtr) calloc(1, sizeof(listNode));
   if (!firstEl)       return NULL;
   firstEl->count = i; lastEl = firstEl; i++;
   while  (i <= n) {
      lastEl->next = (ListPtr) calloc(1, sizeof(listNode));
      if (!lastEl->next)     return NULL;
      lastEl = lastEl->next;
      lastEl->count = i++;
   }
   return firstEl;
}

void printListAsc(ListPtr p) {
   while (p) {
      printf("%d ", p->count);
      printf(" | "); p=p->next;
   }
}

void printListDesc(ListPtr p) {
   if (p != NULL) {
      printListDesc(p->next);
      printf("%d ", p->count); printf(" | ");
   }
}
int main() {
   ListPtr  p;

   if (!(p = buildList(10)))    return 1;
   printf("Liste (von vorne):\n");
   printListAsc(p); printf("\n");
   printf("Liste (von hinten):\n");
   printListDesc(p); printf("\n");
   return 0;
}
```

Kapitel 16

Komplexe Deklaration

Beispiel: Ein Zeiger auf eine Funktion, die einen Zeiger auf einen 5-elementigen Vektor liefert, dessen Elemente Zeiger auf `int`s sind.

```
int *(*(*x)())[5];
```

entziffert von innen nach außen:

`*x`	Zeiger auf
`(*(*x)())`	`()` bindet stärker als `*`, also ist `(*x)()` eine Funktion, die einen Zeiger liefert Also: `x` ist Zeiger auf Funktion, die einen Zeiger liefert
`int *(*(*x)())[5]`	`[]` bindet stärker als `*`, also ist `x` Zeiger auf eine Funktion, die einen Zeiger auf einen Vektor liefert, dessen 5 Elemente Zeiger auf `int`s sind.

Transparenter wird dies durch stufenweisen Aufbau mit **typedef**:

```
typedef int *AP[5];
```

`AP` ist damit Name für einen 5-elementigen Vektor mit Zeigern auf `int`.

```
typedef AP *FP();
```

`FP` ist damit ein Name für eine Funktion, die Zeiger auf `AP` liefert. Und schliesslich

```
FP * x;
```

Beachte:

- Der Vektor-Operator `[]` und der Funktionsoperator `()` haben höheren Vorrang als der Zeigeroperator `*`!
- `[]` und `()` werden von links nach rechts gruppiert, `*` von rechts nach links!

Beispiele:

```
char *x[];
```

- `x[]` ist Vektor
- `*x[]` ist Vektor von Zeigern
- `char *x[]` ist Vektor von Zeigern auf `int`

Klammern ändern den Vorrang:

```
int (*x[])();
```

- `x[]` ist Vektor
- `(*x[])` ist Vektor mit Zeiger-Elementen
- `(*x[])()` ist Vektor mit Zeigern auf Funktionen
- `int (*x[])()` ist Vektor von Zeigern auf Funktionen, die `int` liefern.

ohne Klammern:

```
int*x[]();
```

- ein Vektor von Funktionen, die Zeiger auf `int`s liefern
- **unzulässig**, da Vektoren mit Funktionen als Elemente nicht erlaubt!

Beispiel: Ein Zeiger auf einen Vektor, dessen Elemente Zeiger auf Funktionen sind, die Zeiger auf Vektoren liefern, deren Elemente Strukturen mit *tag*-Name `S` sind:

1.	`(*x)`	`x` ist Zeiger
2.	`(*x)[]`	`x` ist Zeiger auf Vektor
3.	`(*(*x)[])`	`x` ist Zeiger auf Vektor, dessen Elemente Zeiger sind
4.	`(*(*x)[])()`	`x` ist Zeiger auf Vektor, dessen Elemente Zeiger auf Funktionen sind
5.	`(*(*(*x)[])())`	`x` ist Zeiger auf Vektor, dessen Elemente Zeiger auf Funktionen sind, die Zeiger liefern
6.	`(*(*(*x)[])())[]`	`x` ist Zeiger auf Vektor, dessen Elemente Zeiger auf Funktionen sind, die Zeiger auf Vektoren liefern
7.	`struct S (*(*(*x)[])())[]`	`x` ist Zeiger auf Vektor, dessen Elemente Zeiger auf Funktionen sind, die Zeiger auf Vektoren liefern, deren Elemente Strukturen mit *tag*-Name `S` sind

Beispiele für unzulässige Deklarationen:

`int af[]()`	Vektor von Funktionen, die `int`s liefern
`int fa()[]`	Eine Funktion, die einen Vektor mit `int`s liefert
`int ff()()`	Eine Funktion, die eine Funktion liefert, die `int`s liefert
`int (*paf)[]()`	Ein Zeiger auf einen Vektor von Funktionen, die `int`s liefern
`int * ffp()()`	Eine Funktion, die eine Funktion liefert, die einen Zeiger auf `int`s liefert

Kapitel 17

Argumente aus der Kommandozeile

Die Funktion `main` hat zwei Argumente (eigentlich drei): `int argc` und `char **argv`:

- `argc\`
  Diese `int`-Größe gibt die Anzahl der Argumente auf der Kommandozeile (incl. Kommandoname)
- `argv`
  Dies ist ein Zeiger auf einen Vektor mit Zeichenketten, die die einzelnen Argumente enthalten, ein Argument pro Zeichenkette.

Nach Konvention ist `argv[0]` der Kommando-/Programmname, `argc` ist also mindestens 1.

Beispiel: alle Argumente inkl. Programmname ausgeben

```
/* arg1.c */

#include <stdio.h>

void main(int argc,char *argv[]) {
  int i;
  for (i = 0; i<argc; i++)
     printf("%s%c", argv[i], (i < argc-1) ? ' ':'\n');
}
```

Da `argv` ein Zeiger auf einen Vektor von Zeigern ist, gibt es mehrere Möglichkeiten, dieses Programm zu realisieren (jetzt ohne Programmname):
`argv` ist ein Zeiger auf den Anfang des Vektors mit den Kommando-Argumenten; somit sorgt die Inkrementierung `++argv` dafür, das dieser Zeiger auf das Element `argv[1]` zeigt und nicht mehr auf `argv[0]`.

```
/* arg2.c */

#include <stdio.h>

void main(int argc, char *argv[]) {
  while (--argc > 0 )
     printf("%s%c", *++argv, (argc > 1) ? ' ':'\n');
}
```

Oder:

```
/* arg3.c */
#include <stdio.h>
void main(int argc, char *argv[]) {
   while( --argc > 0)
       printf((argc > 1) ? "%s" : "%s\n", *++argv);
}
```

- Ein Pattern-Suchprogramm a la grep:

```
/* mygrep.c */
#include <stdio.h>
#include <strings.h>    /* fuer strchr etc .. */

#define LENGTH  1000

/* liefert Indexposition, ab der t in s vorkommt */
int myindex(char *s, char *t) {
   int tlen = strlen(t);    /* Stringlaenge von t */
   char *p = s;

   while (p = strchr(p, t[0]))
        if (!strncmp(p, t, tlen))  /* gefunden */
                return p - s;
        else                /* weitersuchen   */
                p++;
   return -1;
}

/* Suchmuster als 1.Argument        */
int main(int argc, char *argv[]) {
   char line[LENGTH];

   if (argc != 2) {
       fprintf(stderr, "Usage: %s pattern\n", argv[0]);
       return 1;
   } else
       while (fgets(line,LENGTH,stdin))
           if (myindex(line,argv[1]) >=0)
              printf(line);
   return 0;
}
```

Modifikation dahingehend, dass **zwei** optionale Argumente erlaubt sein sollen: Das eine Argument verlangt „Gib alle Zeilen aus, mit Ausnahme derer, die das Suchmuster enthalten“, das zweite „Gib vor jeder Ausgabezeile die Zeilennummer aus“:
Anm.:

- Statt `(*++argv)[0]` wäre auch `**++argv` möglich!
- Man beachte: `(*++argv)[0]` und `*++argv[0]` sind verschieden: letzteres ist als `*++(argv[0])` zu lesen.

```
/*  mygrep1.c:  -x -n pattern oder  -nx pattern */
#include <stdio.h>
#include <strings.h>    /* fuer strchr etc .. */

#define LENGTH  1000

/* liefert Indexposition, ab der t in s vorkommt */
int myindex(char *s, char *t) {
   int tlen = strlen(t);    /* Stringlaenge von t */
   char *p = s;

   while (p = strchr(p, t[0]))
       if (!strncmp(p, t, tlen))  /* gefunden */
                return p - s;
       else                /* weitersuchen   */
               p++;
   return -1;
}

int main(int argc, char *argv[]) {
    char line[LENGTH], *s;
    long lineno = 0; int except = 0, number = 0;

     /* ueber alle Optionen (Argument beginnt mit - )*/
    while (--argc > 0 && (*++argv)[0] =='-')
      for (s = argv[0] + 1; *s != '\0'; s++)
         switch (*s) {    /* Flags abarbeiten  */
           case 'x': except = 1; break;
           case 'n': number = 1; break;
           default :
              fprintf(stderr, "%s: illegal option >%c<\n",
                      argv[0], *s);
              argc = 0;
         }

    if (argc != 1) {
         fprintf(stderr, "Usage: %s -x -n pattern \n",
                  argv[0]); return 1;
    } else
         while (fgets(line, LENGTH, stdin)) {
            lineno++;
            if ((myindex(line,*argv) >=0) != except) {
               if (number)
                  printf("%ld: ", lineno);
               printf(line);
            }
         }
   return 0;
}
```

Kapitel 18

Mehr zum Preprozessor

18.1 Übersicht

Wichtigste Merkmale:

- Makro-Verarbeitung
- Einfügen von Quelldateien („Modularisierung“)
- bedingte Übersetzung

Preprozessor-Anweisungen sind zeilenorientiert, beginnen mit # (das erste Nicht-Leerzeichen in der Zeile), können an beliebigen Stellen im C-Programm stehen (meist jedoch am Anfang) und enden mit *newline*. Falls mehr als eine Zeile benötigt wird, steht vor dem *newline* ein Backslash:

```
thales$ cat bsp1.c
# define LONG_MACRO "Dies ist ein langes Makro,\
das deutlich ueber eine\
Zeile hinausgeht."

: LONG_MACRO :
thales$ gcc -E bsp1.c
# 1 "bsp1.c"




: "Dies ist ein langes Makro,das deutlich ueber eineZeile hin-
ausgeht."  :
thales$ gcc -E -P bsp1.c


: "Dies ist ein langes Makro,das deutlich ueber eineZeile hin-
ausgeht."  :
thales$
```

18.2 Makro-Substitution

Makro: Ein Name, dem ein Text-String zugeordnet ist (Makro-Rumpf). Nach Konvention werden Makro-Namen, die Konstanten repräsentieren, groß geschrieben (Unterscheidung von Makro- und Variablen-Namen).

Textersatz: jedes Vorkommen eines Makro-Namens außerhalb seiner Definition (Makro-Aufruf) wird vom Preprozessor durch den Makro-Rumpf (Ersatztext) ersetzt, in die Eingabe für den Preprozessor zurückgestellt und ggf. erneut bearbeitet.

18.3 define

```
"#" "define" macro_name ["("macro_argument
                             {"," macro_argument}
                          ")"] macro_body
```

Syntax 18.1: define-Makro

Fehlerquellen:

```
thales$ cat bsp2.c
# define SIZE 10;
# define OKAY (var == 1);
# define MAX = 100
# define neg_a_plus_f(a) -(a) + f
# define neg_b_plus_f (b)  -(b) + f
# define min(a,b) ((a) < (b) ? (a) : (b))

a1) x = SIZE;
a2) int array[SIZE];
b) while OKAY
        tue();
c) for (j=MAX; j>0; j--)
d) j = neg_a_plus_f(x);
e) j = neg_b_plus_f(x);
f) a = min(b++,c);
thales$ gcc -E -P bsp2.c
```

```
a1) x = 10; ;
a2) int array[10; ];
b) while (var == 1);
        tue();
c) for (j== 100 ; j>0; j--)
d) j = -( x ) + f ;
e) j = (b)  -(b) + f (x);
f) a = (( b++ ) < ( c ) ? ( b++ ) : ( c )) ;
```

```
thales$
```

a1) kein Problem, da `;;` ein leeres Statement ist!

a2) wird vom Compiler entdeckt

b) hat die Variable `var` tatsächlich den Wert 1, so Endlosschleife

c) wohl nicht gewollt!

d) okay

e) was, wenn `b` eine vereinbarte Variable ist?

f) Seiteneffekt: `b` wird u.U. zweimal inkrementiert!

Beispiel:

```
thales$ cat bsp3.c
# define square(a) a*a
# define quadrat(a) (a)*(a)

j = 2 * square(3+4)
i = 2 * quadrat(3+4)

thales$ gcc -E -P bsp3.c



j = 2 *  3+4 * 3+4
i = 2 * ( 3+4 )*( 3+4 )
thales$
```

- Makro-Argumente haben keinen Typ, also keine Typprüfung
- Makro's sind "schneller" als Funktionen (wirklich ein Argument für Makro's?)

18.4 Entfernen von Makro-Definitionen

```
"#" "undef" macro_name
```

Syntax 18.2: undef-Makro

Beispiel:

```
thales$ cat bsp4.c
# define NAME Schweiggert
Herr NAME ist Leiter der SAI.
# undef NAME
# define NAME Borchert
Herr NAME ist Mitarbeiter der SAI
thales$ gcc -E -P bsp4.c
```

```
Herr Schweiggert  ist Leiter der SAI.


Herr Borchert  ist Mitarbeiter der SAI
thales$
```

18.5 built-in-Makro's

```
thales$ cat bsp5.c
# include <stdio.h>

# define CHECK(a,b) \
  if ((a) != (b))  \
        fail(a,b, __FILE__ , __LINE__ )

void fail(int a, int b, char *p, int line)
{
        printf("Check failed in file %s at line %d:\n\
        received %d, exspected %d\n", p, line, a, b);
}

main()
{
        int x = 1;
        CHECK(x,0);
}
thales$ gcc bsp5.c
thales$ a.out
Check failed in file bsp5.c at line 16:
        received 1, exspected 0
thales$
```

18.6 Bedingte Übersetzung

```
"#" "if" cond_expr
C-source-code
{ "#" "elif" cond_expr
C-source-code
}
[ "#" "else"
C-Source-code
]
"#" "endif"
```

Syntax 18.3: if-Makro

Anmerkung: das `elif`-Konstrukt ist ANSI (also nicht in allen K&R-Compilern verfügbar)!

Beispiel:

```
thales$ cat bsp6.c
# define x 2
# if x == 1
# undef x
# define x 0
# elif x == 2
# undef x
# define x 3
# else
# define y 4
# endif
a) xWert ist x
b) yWert ist y

thales$ gcc -E -P bsp6.c




a) xWert ist 3
b) yWert ist y
thales$
```

Anmerkung: der `cond_expr` **muß** ein konstanter Ausdruck sein, d.h. in obigem Beispiel muß `x` ein Makro sein. Außerdem werden dort Makros erst expandiert und dann wird der Ausdruck bewertet. Nicht definierte Namen werden meist durch 0 ersetzt (oder Fehler).

Beispiel:

```
thales$ cat bsp7.c
# include <stdio.h>
```

```
main()
{
        int i = 9;
        int j = 10;

# if DEBUG
        printf(" Wert von i / j: %d / %d\n", i,j);
# endif

        printf("Ergebnis: %d\n", i*j);
}
thales$ gcc bsp7.c
thales$ a.out
Ergebnis: 90
thales$ gcc -DDEBUG=1 bsp7.c
thales$ a.out
 Wert von i / j: 9 / 10
Ergebnis: 90
thales$
```

Anmerkung: die Option `-Dname=wert` ist äquivalent zur Makro-Definition `#define name wert`.

Bei der Übergabe von Doppelanführungszeichen auf Kommandoebene muß wegen derer Interpretation durch die Shell eine Klammerung in einfache Hochkommas erfolgen:

```
$cc -DZAHL=float -D'INFMT="%f"' -D'OUTFMT="%g"' ...
```

18.7 Test auf Makro-Existenz

ifdef

Beispiel:

```
thales$ cat bsp8.c
# include <stdio.h>

# ifndef ZAHL
#       define ZAHL int
#endif

#ifndef OUTFMT
#       define OUTFMT "%d"
#endif

main()
{
        ZAHL i = 9;
        ZAHL j = 10;

# ifdef DEBUG
        printf(" Wert von i / j:");
        printf(OUTFMT,i); printf(OUTFMT, j);
```

```
printf"\n");
# endif

        printf("Ergebnis:"); printf(OUTFMT, i*j);
printf("\n");
}
thales$ gcc -DZAHL=float -DOUTFMT='"%f"' bsp8.c
thales$ a.out
Ergebnis:90.000000
thales$ gcc bsp8.c
thales$ a.out
Ergebnis:90
thales$
```

18.8 include

- `# include <filename>`
- `# include "filename"`

Im ersten Fall sucht der Preprozessor an bestimmten Stellen (i.a. `/usr/include`, bei manchen Compilern muss auch die Umgebungsvariable `INCLUDE` auf den Pfad gesetzt sein) nach der Datei `filename` und setzt deren Inhalt an die Stelle von `include`.
Im zweiten Fall wird im eingestellten Pfad gesucht; wenn dort nicht enthalten, so Suche wie bei „`< >`“.

Verwendung: Modularisierung (s.u.)

Kapitel 19

Speicherklassen

C-Programme können aus verschiedenen Quelldateien bestehen, die nacheinander übersetzt werden. Ebenso können Funktionen aus Büchereien geladen werden.

19.1 Geltungsbereich

Der Geltungsbereich eines Namens ist der Teil des Programms, für den er vereinbart wurde. Variable, die am Anfang einer Funktion vereinbart wurden (Speicherklasse **auto**), haben den Geltungsbereich der Funktion.

Der Geltungsbereich von Variablen, die außerhalb einer Funktion, also **extern** vereinbart wurden, ist von der Stelle der Vereinbarung bis zum Ende dieser Datei. Soll eine externe Variable verwendet werden, **bevor** sie in einer Quelldatei definiert wird oder wird eine Variable verwendet, die in einer **anderen** Quelldatei definiert ist, so ist eine **extern**-Deklaration notwendig (und aus Lesbarkeitsgründen sinnvoll)!

19.2 Deklaration vs. Definition

- Deklaration:
  legt Eigenschaften einer Variable (Typ, Speicherbedarf, u.a.) fest (Bekanntmachen).
- Definition:
  sorgt zusätzlich dafür, dass Speicherplatz bereitgestellt wird.

Beispiel:

```
int vekt[10];
int i=0;

int main() {
...
}
```

Hier werden zwei Variablen `vekt` und `i` definiert, es wird Speicherplatz bereitgestellt und sie sind deklariert.

```
# include <stdio.h>

extern int vekt[];
```

```
extern int i;

main() {
...
}
```

Hier werden zwei Variablen `vekt` und `i` für den Rest der Datei deklariert, die Variablen werden nicht „erzeugt“, es wird kein Speicherplatz angelegt, sie sind an anderer Stelle definiert.

- Es darf nur **eine** Definition in allen Quelldateien zu einem Programm geben!
- Andere Dateien, die diese Variable benötigen, **müssen** diese mit einer `extern`-Deklaration bekannt machen!
- Initialisierung einer `extern`-Variable kann nur bei der Definition erfolgen!

Beispiele:

```
thales$ cat file.c
float c = 4.5;

thales$ cat main0.c

main() {
        printf("Wert von c: %f\n", c);
}

thales$ gcc file.c main0.c
main0.c: In function ‘main’:
main0.c:3: ‘c’ undeclared (first use this function)
main0.c:3: (Each undeclared identifier is reported only once
main0.c:3: for each function it appears in.)
```

Oder besser so?

```
thales$ cat file.c
float c = 4.5; /*Definition*/

titania$ cat main1.c
extern c;

main() {
printf("Wert von c: %f\n", c);
}

thales$ gcc file.c main1.c
thales$ a.out
Wert von c: 1024.000000
```

Oder besser so?

```
thales$ cat file.c
float c = 4.5; /*Definition*/

thales$ cat main2.c

float c;

main() {
printf("Wert von c: %f\n", c);
}

thales$ gcc file.c main2.c
thales$ a.out
Wert von c: 4.500000
```

Oder besser so?

```
thales$ cat file.c
float c = 4.5; /*Definition*/

thales$ cat main3.c
extern float c;

main() {
printf("Wert von c: %f\n", c);
}

thales$ gcc file.c main3.c
thales$ a.out
Wert von c: 4.500000
```

19.3 static

Variablen, die innerhalb einer Funktion mit vorangestelltem `static` definiert werden, haben eine Lebensdauer, die über die der Funktion hinausgeht; sie werden beim ersten Aufruf erzeugt und behalten ihren Wert auch nach Bendigung der Funktion.

Variablen, die außerhalb von Funktionen definiert werden, können mit dem vorangestellten `static` in ihrem Gültigkeitsbereich auf die Datei beschränkt werden. Auch Funktionen kann ein `static` vorangestellt werden, mit derselben Wirkung.

Kapitel 20

Modularisierung

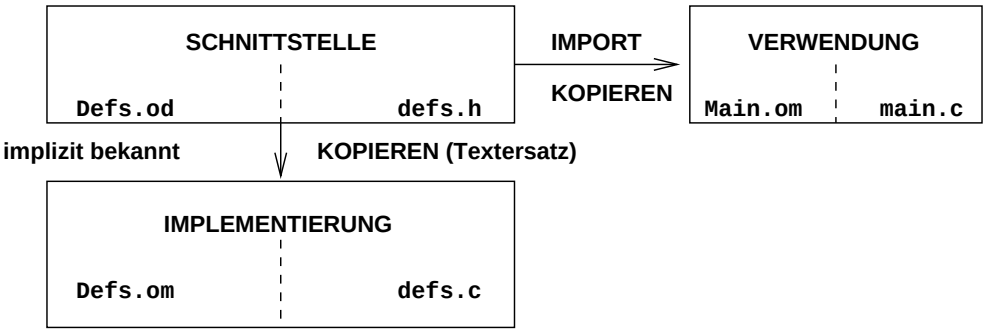

Abbildung 20.1: IMPORT vs. INCLUDE

Entsprechend dem Prinzip in Modula-2 / Oberon werden auch in C größere Programme in „Teile“ (Dateien!) zerlegt, wobei das Prinzip *Information Hiding* (in Modula-2 mit *DEFINITION MODULE — IMPLEMENTATION MODULE*, in Oberon mit *DEFINITION — MODULE*) zugrundegelegt werden sollte.

Zur Definition der Schnittstelle (*DEFINITION*) werden in C **header files** (`<name>.h`) verwendet, die per **include** „importiert“ (besser: bekannt gemacht) werden (man beachte die Wirkung von *include*). In der Datei `<name>.c` erfolgt die zugehörige Implementierung. Hierbei kann es allerdings zu sehr unliebsamen (teilweise compilerabhängigen) Effekten kommen.

Beispiel (in Oberon):

```
DEFINITION Defs;
   CONST
      Factor = 1331;
      Upper = 100;
      Lower = 1;
   TYPE
      RandomNumber = INTEGER;
   VAR
      start: RandomNumber;

   PROCEDURE Rand(VAR s: RandomNumber):RandomNumber;

END Defs.
```

```
MODULE Defs;

   PROCEDURE Rand(VAR s: RandomNumber):RandomNumber;
   BEGIN
      s := Factor*start MOD Upper + Lower;
      start := s;
      RETURN s
   END Rand;
BEGIN
   start := 1
END Defs.
```

```
DEFINITION Main;
END Main.
```

```
MODULE Main;
   IMPORT Defs, Write;

   VAR s: Defs.RandomNumber;
BEGIN
   Defs.start := 2;
   Write.Int(Defs.Rand(s),5);
   Write.Ln
END Main.
```

Die erste Umsetzung in C könnte wie folgt aussehen:

```
/* Defs.h */
# define Factor 1331
# define Upper 100
# define Lower 1

typedef int RandomNumber;

RandomNumber start = 1;
RandomNumber Rand(RandomNumber * s);



/* defs.c */
# include "defs.h"

RandomNumber Rand(RandomNumber * s) {
    *s = ((Factor*start) % Upper) + Lower;
        start = *s;
        return *s;
}



/* main.c */
# include "defs.h"
void main() {
    RandomNumber s;

    start = 2;
    printf("in C: %d\n", Rand(&s));
}
```

Einfach übersetzt:

```
thales$ gcc defs.c main.c
ld: fatal: symbol ‘start’ is multiply defined:
(file /tmp/cca001er1.o and file /tmp/cca001er2.o);
ld: fatal: File processing errors.  No output written to a.out
```

So geht es also nicht!

Das „Kopieren“ des Header-Files muß unterschiedlich erfolgen:

```
/* Defs.h */
# ifdef defs_h
# define EXTERN
# define INIT = 2
# else
# define EXTERN extern
# define INIT
#endif

# define Factor 1331
# define Upper 100
# define Lower 1

typedef int RandomNumber;

EXTERN RandomNumber start INIT;

EXTERN RandomNumber Rand(RandomNumber * s);
```

```
/* defs1.c */
#define defs_h
# include "defs1.h"


RandomNumber Rand(RandomNumber * s) {
    *s = ((Factor*start) % Upper) + Lower;
        start = *s;
        return *s;
}
```

```
/* main1.c */
# include "defs1.h"

main() {
    RandomNumber s = 2;
    start = 10;
    printf("in C: %d\n", Rand(&s));
    printf("in C: %d\n", Rand(&s));
}
```

Übersetzung:

```
thales$ gcc -E defs.c main.c
# 1 "defs.c"
# 1 "defs.h" 1




typedef int RandomNumber;

RandomNumber start = 1;
RandomNumber Rand(RandomNumber * s);
# 1 "defs.c" 2


RandomNumber Rand(RandomNumber * s) {
*s = ((1331 *start) % 100 ) + 1 ;
start = *s;
return *s;
}

# 1 "main.c"
# 1 "defs.h" 1




typedef int RandomNumber;

RandomNumber start = 1;
RandomNumber Rand(RandomNumber * s);
# 1 "main.c" 2


main() {
RandomNumber s;
printf("in C: %d\n", Rand(&s));
}
```

Kapitel 21

Abstrakte Datentypen (ADTs)

21.1 Prinzip

Nur der Typ-Name sowie die Operationen (Prozedur-Köpfe — Prototypen der Prozeduren) darauf werden an der Schnittstelle bekanntgegeben, die Implementierung (Typ-Struktur wie Prozedur-Rümpfe) selbst ist verborgen!

```
/* ---------- adt.h ------------*/

typedef struct _object * Object; /*ADT*/

typedef int BOOL;

/* Operationen: */
BOOL init(Object * x);
BOOL square(Object * x);

void printObjValue(Object x);
```

```
/* -------------- adt.c ----------------------------- */
# include <stdio.h>
# include <stdlib.h>
# include "adt.h"

struct _object {
  int cont;
  float value;
} ;

BOOL init(Object * x) {
  return (*x = (Object) calloc(1,sizeof(struct _object)))
      && ( (*x)->cont = 1 ) && ( (*x)->value=100);
}

BOOL square(Object * x) {
   (*x)->value *= (*x)->value;
   return 1;
}

void printObjValue(Object x) {
  printf("%f\n", x->value);
}
```

```
/* ------------------ mainAdt.c ------------- */
# include "adt.h"

Object x;
int main() {
   if ( init(&x) )
      printObjValue(x);
   if (square(&x) )
      printObjValue(x);
   exit(0);
}
```

21.2 Abstrakter Datentyp „Stack“

```
/*
 *    stack.h - module interface for stack.c
 */

# ifndef STACK_H
# define EXTERN_H extern
# else
# define EXTERN_H
#endif

# ifdef ANSI_C
#       define VOID void
# else
#       define VOID char
# endif

typedef VOID *ELEMENT;
/* type of data to be stored    */
typedef struct _stack * STACK;
/* ADT STACK (hidden type)      */
typedef int BOOL;
/* to indicate boolean results  */
typedef ELEMENT (*COPYPROC)(/*ELEMENT elem */);
        /* copy procedures              */

# ifdef ANSI_C
BOOL create (STACK *sp,  COPYPROC copy);
BOOL delete (STACK stack);
BOOL push   (STACK stack, ELEMENT elem);
BOOL pop    (STACK stack, ELEMENT *ep);
# else
EXTERN_H BOOL create ( /* STACK *sp,  COPYPROC copy */ );
EXTERN_H BOOL delete ( /* STACK stack */ );
EXTERN_H BOOL push   ( /* STACK stack, ELEMENT elem */ );
EXTERN_H BOOL pop    ( /* STACK stack, ELEMENT *ep  */ );
# endif
```

```
/* stack.c */
# include <stdlib.h>
#define STACK_H
#include "stack.h"

typedef struct _node *NODE;

#define TRUE  1
#define FALSE 0

struct _stack {
        COPYPROC copy;
        NODE top;
};
struct _node {
        ELEMENT elem;
        NODE next;
};
/* Stack erzeugen & Copyproc einhaengen */
BOOL create (STACK *sp, COPYPROC copy) {
       return (*sp = (STACK) calloc(1,sizeof(struct _stack)))
                 && ((*sp)->copy = copy);
}
/* Stack freigeben, wenn existent und leer */
BOOL delete (STACK stack) {
       if (!stack || stack->top) return FALSE;
       else {
                free(stack);
                return TRUE;
        }
}
/* Element auf den Stack legen */
BOOL push(STACK stack, ELEMENT elem) {
       NODE node;
        if (stack && (node = (NODE) cal-
loc(1,sizeof(struct _node)))
            && (node->elem = (*stack->copy)(elem))) {
           node->next = stack->top;
           stack->top = node;
           return TRUE;
        } else return FALSE;
}
/* zuletzt eingefuegtes Element vom Stack holen */
BOOL pop(STACK stack, ELEMENT *elem) {
        if (stack && stack->top) {
          NODE node = stack->top;
          stack->top = node->next;
          *elem = node->elem;
          free(node);
          return TRUE;
        } else return FALSE;
}
```

```
/* stack - main.c */
# include <string.h>    /* fuer strdup */
# include <stdio.h>
# include "stack.h"

# define MAXLINE 80

int main() {
  char line[MAXLINE];
  char *out;
  STACK stack;

  puts("Ein Stack fuer Zeichenketten!");
  puts("Erstes Zeichen gibt Operation:");
  puts("+element: Push");
  puts("-       : Pop (incl. Top");
  puts("*       : Empty");
  puts("!       : Quit");

  if (!create(&stack, strdup)) /* strdup legt Stringko-
pie an! */
       return 1;               /* Stackanle-
gen ging schief    */

  while (fgets(line,MAXLINE,stdin)) {
       line[strlen(line)-1] = '\0'; /*newline von fgets()*/
       switch(line[0]) {
          case '+':
                printf("pushed (%d): >%s<\n",
                        push(stack,line+1), line+1);
                break;
          case '-':
                if (pop(stack,&out)) {
                    printf("popped: >%s<\n", out);
                }
                else
                    printf("pop failed (stack empty?)\n");
                break;
          case '*':
                while (pop(stack,&out))
                    printf("popped: >%s<\n",out);
                break;
          case '!': return 0;

          default:
                puts("expected [-+*!] as first char");
        }
      }
      printf("deleting stack:%d \n", delete(stack));
      return 0;
}
```

Kapitel 22

Einige Tools

22.1 Compiler

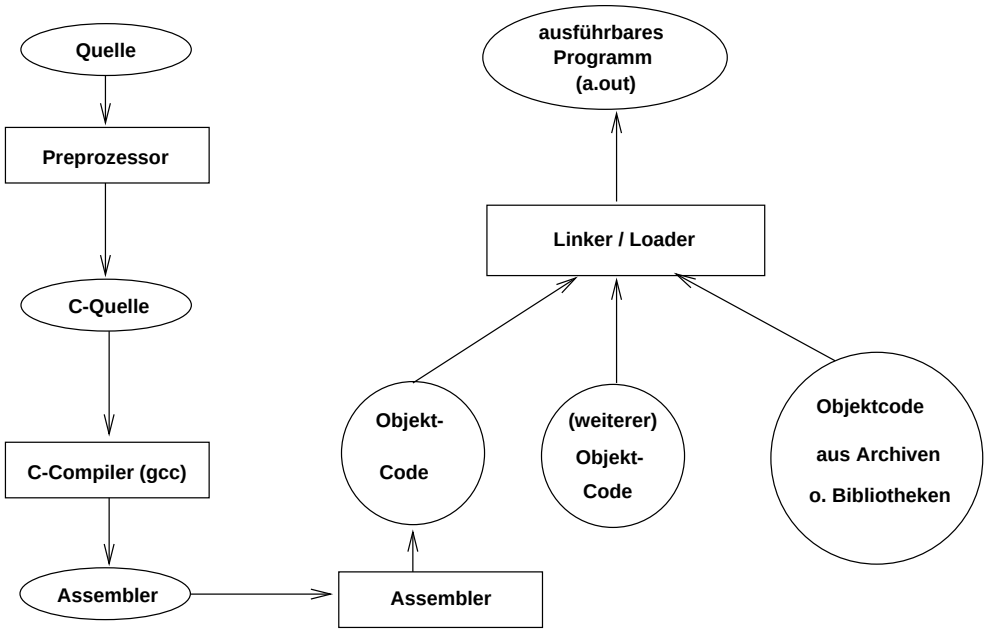

Abbildung 22.1: Compiler-Phasen

Der **C-Compiler** arbeitet in mehreren Durchgängen:

- Zwischenräume, Kommentare entfernen und TOKEN bilden
- Assembler-Text erzeugen

```
hypatia$ cat trivial.c /* Intel-Prozessor */
/*trivial.c - ein trivial Programm*/

#include <stdio.h>
#include "mul.h"
```

```
int main() {
int i, j;
i= 2; j=4;
i = mul(i,j);
printf("Ergebnis: \%d\n", i);
exit(0);
}
hypatia$ gcc -S trivial.c
hypatia$ cat trivial.s
.file "trivial.c"
.version "01.01"
gcc2_compiled.:
.section .rodata
.LC0:
.string "Ergebnis: %d\n"
.text
.align 16
.globl main
.type  main,@function
main:
pushl %ebp
movl %esp,%ebp
subl $8,%esp
movl $2,-4(%ebp)
movl $4,-8(%ebp)
movl -8(%ebp),%eax
pushl %eax
movl -4(%ebp),%eax
pushl %eax
call mul
addl $8,%esp
movl %eax,%eax
movl %eax,-4(%ebp)
movl -4(%ebp),%eax
pushl %eax
pushl $.LC0
call printf
addl $8,%esp
pushl $0
call exit
addl $4,%esp
.align 16
.L1:
movl %ebp,%esp
popl %ebp
ret
.Lfe1:
.size  main,.Lfe1-main
.ident "GCC: (GNU) 2.7.2.1"
hypatia$
```

Der **Assembler** erzeugt aus dem Assembler-Code den Object-Code (Maschinensprache).

Der **Linker/Loader** (**ld**)

- sucht in seinen Argumenten nach Objektdateien und Bibliotheksdateien
- wenn gefunden, so liest er diese Datei, verschiebt den Programmcode und fügt dieses Programm am Ende der binären ausführbaren Datei an
- mischt die Symboltabelle des Objektprogramms mit der des binären ausführbaren Programms

22.2 Archivdateien

(siehe `/lib` resp. `/usr/lib`)

Archivdateien haben die Endung „`.a`“.

- Der **Linker/Loader** durchsucht die Symboltabelle des Archivs, um Namen zu finden, die undefinierte Namen in der Symboltabelle der binären ausführbaren Datei auflösen.
- diejenigen Objektprogramme des Archivs, die Namen „auflösen“, werden zur ausführbaren Datei hinzugefügt.

Der Linker/Loader (**ld**) wird typischerweise indirekt über den C-Compiler aufgerufen:

```
hypatia$ cat math.c
# include <stdio.h>

extern double sin();

void main() {
double x;

printf("sin(%f) = %f\n",x, sin(x));
}
hypatia$ gcc -Wall math.c

/tmp/cca011561.o: In function ‘main’:
/tmp/cca011561.o(.text+0xd): undefined reference to ‘sin’

hypatia$ gcc -Wall math.c -lm # libm.a

hypatia$ a.out
sin(0.000000) = 0.000000
hypatia$
```

Häufig verwendete Bibliotheken (Archivdateien):

- **libc.a**: C-Standardbibliothek
- **libm.a**: mathematische Funktionsbibliothek
- **libcurses.a**: Bildschirmsteuerung

Eigene Bibliotheken:

Kommando **ar** — mehr dazu in den Manualseiten zu `ar` (`man ar`)!

22.3 make

- *make* ist ein Werkzeug zur Verwaltung mehrerer, miteinander in „Beziehung“ stehender Dateien.
- In Zusammenhang mit Modula-2 / Oberon und dem Kommando *mmm* (*make makefile for modula*) / *mmo* (*make makefile for oberon*) wurde *make* zur effizienten Übersetzung von Modula-2- / Oberon-Modulen benutzt.
- Die von *make* interpretierte Steuerdatei **makefile** muß bei C-Programmen selbst erstellt werden! Der Name kann auch anders gewählt werden und muß dann bei *make* als Argument angegeben werden (`make -f my_makefile` — siehe `man make`)

Aufbau der Steuerdatei makefile:

Ein *makefile* enthält Einträge der Form

```
Target: <TAB> {<TAB>} {Dependant}
<TAB> {<TAB>} Command
<TAB> {<TAB>} Command
        .
        .
        .
```

Target und *Dependant* sind in der Regel Dateinamen, *Command* ein beliebiges UNIX-Kommando / ausführbares Programm. Das Zeichen # leitet einen Kommentar bis zum Zeilenende ein.

Beispiel:

```
/* main.h */
#define LINE_SIZE  80


/* eingabe.h */
extern int eingabe();


/* ausgabe.h */
extern void ausgabe();
```

```
/*  main.c
 */
#include <stdio.h>
#include "main.h"
#include "eingabe.h"
#include "ausgabe.h"

int main() {      /* Hauptprogramm */
char line[LINE_SIZE];
   while (eingabe(line))
      ausgabe(line);
   putchar('\n');
   return 0;
}
```

```
/* eingabe.c
 *  Liest eine Zeile von stdin und schreibt
 *  sie in den uebergebenen String.
 *  Returnwert : die Laenge des eingegebenen
 *               Strings.
 */

#include <stdio.h>

int eingabe(char *buffer) {
char * get_buf;
   printf("\t Eingabe : ");
   get_buf = gets(buffer);
   return ( get_buf == NULL ? 0 : strlen(get_buf));
}
```

```
/* ausgabe..c
 *  Gibt eine Zeile in Grossbuchstaben am Bildschirm aus.
 */
#include <stdio.h>

char * upper();   /* Funktionsprototyping */

void ausgabe(char *buffer) {
   printf("\t Ausgabe : %s \n\n", upper(buffer));
}

/* String-Konvertierung in Grossbuchstaben */
char *upper(char *string) {
   char *pos; int offset;

   offset = 'a' - 'A';
   for (pos = string; *pos; pos++)
      if (*pos >= 'a' && *pos <= 'z')
         *pos = *pos - offset;

   return string;
}
```

Zur effizienten Übersetzung dieser drei C-Programme kann die folgende Steuerdatei verwendet werden:

```
# MAKEFILE --- Version 1
main:       main.o eingabe.o ausgabe.o
            gcc -Wall -o main main.o eingabe.o ausgabe.o
main.o:            main.c main.h eingabe.h ausgabe.h
            gcc -Wall -c main.c
eingabe.o:      eingabe.c eingabe.h
            gcc -Wall -c eingabe.c
ausgabe.o:      ausgabe.c ausgabe.h
            gcc -Wall -c ausgabe.c
clean:
            rm -f main.o ausgabe.o eingabe.o core
realclean:
            rm -f main.o ausgabe.o eingabe.o main core
```

make erzeugt aus dieser Beschreibung von Abhängigkeiten folgende Baumstruktur:

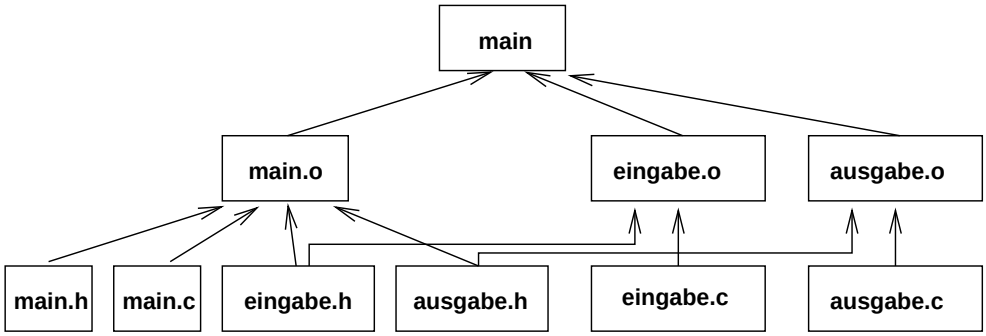

Abbildung 22.2: Abhängigkeiten-Baum

Dieser Baum wird in *postorder* traversiert. Wird dabei festgestellt, dass mindestens eine der Nachfolger-Dateien jünger ist als ihre Vaterknoten-Datei oder dass die Vaterknoten-Datei nicht existiert, so werden die dem Vaterknoten zugeordneten Kommandos ausgeführt.

```
hypatia$ ls
ausgabe.c   eingabe.c   main.c      makefile
ausgabe.h   eingabe.h   main.h
hypatia$ make
gcc -Wall -c main.c
gcc -Wall -c eingabe.c
gcc -Wall -c ausgabe.c
gcc -Wall -o main main.o eingabe.o ausgabe.o
hypatia$ ls
ausgabe.c   ausgabe.o   eingabe.h   main        main.h      makefile
ausgabe.h   eingabe.c   eingabe.o   main.c      main.o
hypatia$ make
make: ‘main’ is up to date.
hypatia$ touch main.h
hypatia$ make
gcc -Wall -c main.c
gcc -Wall -o main main.o eingabe.o ausgabe.o
hypatia$ ls
ausgabe.c   ausgabe.o   eingabe.h   main        main.h      makefile
ausgabe.h   eingabe.c   eingabe.o   main.c      main.o
hypatia$ make clean
rm -f main.o ausgabe.o eingabe.o core
hypatia$ ls
ausgabe.c   eingabe.c   main        main.h
ausgabe.h   eingabe.h   main.c      makefile
hypatia$ make
gcc -Wall -c main.c
gcc -Wall -c eingabe.c
gcc -Wall -c ausgabe.c
gcc -Wall -o main main.o eingabe.o ausgabe.o
hypatia$ make realclean
rm -f main.o ausgabe.o eingabe.o main core
hypatia$ ls
```

```
ausgabe.c   eingabe.c   main.c      makefile
ausgabe.h   eingabe.h   main.h
hypatia$
```

Anmerkungen:

- Wird **make** ohne Parameter aufgerufen, so beginnt die Abarbeitung der Steuerdatei mit dem ersten *target*; soll die Abarbeitung mit einem anderen *target* beginnen, so muß dieses als Parameter mitgegeben werden (*make clean* oder *make realclean* in obigem Beispiel).
- Heißt die Steuerdatei nicht *makefile* bzw. *Makefile*, so muß sie mit der Option `-f` an *make* gegeben werden.
- Die Option `-n` veranlaßt *make*, die resultierenden Befehle am Bildschirm auszugeben, ohne sie jedoch auszuführen
- Man kann im *makefile* Makros definieren und bereits vordefinierte Makros benutzen, um das *makefile* leichter modifizierbar und kompatibler zu gestalten (s.u.).
- *make* geht davon aus, dass eine **.o**-Datei von der zugehörigen **.c**-Datei abhängt und mit dem C-Compiler aus ihr erzeugt wird; diese Abhängigkeiten müssen also nicht explizit im *makefile* aufgeführt werden.
- Ein *target* muß nicht unbedingt ein Dateiname sein, und die Liste der *dependants* kann auch leer sein. Dann werden die zugehörigen Kommandos beim Erreichen des *targets* ausgeführt (siehe im Beispiel *clean*, *realclean*).

```
# MAKEFILE --- Version 2
MAIN= main
EIN=eingabe
AUS=ausgabe
OBJ=$(MAIN).o $(EIN).o $(AUS).o

main:     $(OBJ)
          $(CC) -o $(MAIN) $(OBJ)

main.o:   $(MAIN).h $(EIN).h $(AUS).h
          $(CC) -c $(MAIN).c

clean:
          rm -f $(OBJ) core

realclean:
          rm -f $(OBJ) $(MAIN) core
```

Abbildungsverzeichnis

Aufruf Syntax

Beispielprogramme

Index