Einführung in die Programmiersprache C

1. Übersicht:

Joachim Backes

joachim.backes [at] rhrk.uni-kl.de

http://www.rhrk.uni-kl.de/~backes

Literatur:

1. Kernighan/Ritchie. **Programmieren in C**.
   2. Ausgabe ANSI-C.
   Carl Hanser-Verlag, 1990
   ISBN 3-446-15497-3

2. **Programmiersprache C**. Ein Nachschlagewerk.
   Uni Hannover, RRZN, 11. Auflage 1999

3. Van der Linden. **Expert C Programming. Deep C Secrets.**
   Sun Soft Press, 1994
   ISBN 0-13-177429-8

2. Einstieg

Entwicklung von C eng mit der UNIX-Entwicklung verbunden.

Maßgeblich beteiligt:

Ken Thompson und Dennis Ritchie

C-Werdegang:

- Programmiersprache BCPL: typenfrei
- Programmiersprache B, Nachfolger von BCPL
- 1978: C-Beschreibung in dem Buch

  The C Programming Language

  von Brian Kernighan und Dennis Ritchie, für lange Zeit ein Quasi-Standard (daher auch K&R-C).

Verfügbarkeit:

Verschiedenste Systeme, vom Micro- bis zum Supercomputer.

Standardisierung:

1988 Schaffung des C-Standards ANSI-C, dokumentiert in einer Neuauflage des K&R-Buches.

Wie erstellt man ein Programm:

- Erstellen der Programm-Sourcen mit einem Datei–Editor: ein Programm kann sich über mehrere Dateien erstrecken.
- Übersetzung der Sourcen mit einem C-Compiler [mit automatischem Präprozessor-Aufruf]. Dabei werden (bei fehlerfreien Sourcen) Objekt–Dateien erstellt (Objektmoduln).
- Binden der Objektmoduln zusammen mit (System–)Moduln durch den Linker zu einem ablauffähigen Programm.
- Ausführung des ablauffähigen Programmes durch das Betriebssystem.

Basis-Elemente eines C-Programmes

- Funktionen
- Funktions-Rümpfe und Blöcke
- Typen
- Anweisungen
- Bibliotheksfunktionen

Beispiel

1) Erstelle Datei **`test.c`** mit folgendem Inhalt:

```
#include <stdio.h>                /* Einfügen der Standardbibliothek stdio.h */
main()                            /* Durch main wird das Hauptptogramm markiert */
{
    int i;                        /* Deklariere integer-Variable mit dem Namen i */
    i = 1;                        /* Setze i auf den Wert 1 */
    printf ("i = %d\n", i);       /* Standard-Funktion printf aufrufen */
                                  /* stdout (Bildschirm) ausgeben */
}                                 /* Programmende */
```

2) cc test.c # Uebersetze und binde test.c

3) ./a.out # Das Programm a.out starten

4) Ausgabe: `i = 1`

Sprachcharakteristika anhand des Beispiels

- **/*** und ***/** begrenzen Kommentare
- Vorhandene Formatierung nicht erforderlich! Möglich wäre auch:

```
#include <stdio.h>
main(){int i;i = 1;printf ("i = %d\n", i);}
```

  Aber: **Übersichtlichkeit** geht verloren!

- C-Programme bestehen aus Funktionen, genauer: aus **genau einer** main-Funktion sowie **beliebig vielen anderen** Funktionen.

- Funktionen sind durch Klammernpaare "**{}**" in Blöcke eingeteilt, auch verschachtelt. Der **äußere Block** heißt auch **Funktionsrumpf**.
- Blockinhalte: Anweisungen, jeweils durch Semikolon ";" abgeschlossen. “**int i**” definiert die Integer-Variable i: alle verwendeten Variablen müssen in C deklariert werden mit Typ und Name.

- Die **printf**-Anweisung ist ein Standard-Funktionsaufruf (entnommen aus einem Pool vorübersetzter Funktionen, gehört zur C-Implementierung dazu), die Funktion **printf** wird mit dem Linker dazugebunden.
  Achtung: C selbst kennt als Sprache keine speziellen I/O-Anweisungen !!
- Alle Funktionen sind zu deklarieren, auch Standardfunktionen. Für Standard-funktionen gibt es Header-Dateien, die die Deklaration vornehmen. Das Einbinden von Header-Dateien erfolgt durch den C-Präprozessor mit der Anweisung **#include.**
- Nach Definitionen sind Deklarationen nicht erforderlich.
- Übersetzt wird unter UNIX mit dem Kommando cc:

  Der Kommandoaufruf

  cc *file-1.c* [*file-2.c* ...] [-o *pd*]

  übersetzt die Dateien ***file-1.c*, *file-2.c*,...** in entsprechende Objekt-Dateien und bindet sie zur Programmdatei *pd* bzw. a.out zusammmen.

2. Grundelemente

Zeichensatz

Erlaubte Zeichen sind:

A-Z
a-z
0-9
<Blank>, <HT> und <NL>, <VT>, <FF>
( ) [ ] { } < > + - * / % ^ ~ & | = ! ? # \ , . ;
’ “

durch sog. **Trigraphs:**

??<*ersatzzeichen*>, z.B. entspricht **??**=dem Zeichen **#**

Innerhalb von Zeichenfolgen oder Zeichenkonstanten sind zusätzlich (fast alle übrigen!) weitere Zeichen zulässig, wie z.B. $ oder @.

Formatierung

C-Programme sind **formatfrei**. Sprach-Grundelemente werden durch ein oder mehrere **Trennzeichen** wie z.B. Blank oder HT getrennt.

\ am Zeilenende hat eine Sonderfunktion: Das **auf \ folgende Zeilenende** wird **eliminiert.**

Beispiel:

```
Diese beiden Zeilen \
werden als eine logische Zeile behandelt.
```

Kommentare

Sie dienen der besseren Lesbarkeit und können überall dort stehen, wo Trennzeichen stehen dürfen.

Einleitung durch die Zeichenfolge “**/***”, Beendigung durch “***/**”. Schachtelungen sind nicht möglich!

```
/* Dies ist /* ein Kommentar */
```

Operatoren

Operatoren dienen der Formulierung von Ausdrücken. Manche sind mit mehreren Bedeutungen belegt.

[ ]	()										
.	->	?	:	'							
++	--	&	*	+	-	/	%	~	!	sizeof	
<<	>>	<	>	<=	>=	==	!=	^	|	&&	||
*=	/=	%=	-=	<<=	>>=	&=	^=	|=	+=		
,	=	;	...								
#	##										

Schlüsselwörter

Schlüsselwörter haben eine **syntaktische Bedeutung**. Sie können **nicht anderweitig verwendet** werden.

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

Bezeichner

Durch **Bezeichner** werden die **Objekte** eines Programmes **eindeutig identifiziert** (Objekten sind **stets Speicherstellen** zugeordnet).

Regeln für Bezeichner:

- Folge von Buchstaben, Ziffern und dem Zeichen “_”. Das erste Zeichen darf keine Ziffer sein.
- Die Länge ist beliebig, die Signifikanz ist aber i.A. begrenzt.
- Die Verwendung von Schlüsselwörtern als Bezeichner ist verboten.

Bezeichner sind **Namen** für Variablen, Funktionen, Sprungmarken usf.

Unterscheidung zwischen **internen** und **externen** Namen:

- externe Namen können von **verschiedenen Source–Dateien** eines Programmes angesprochen werden. Mindestens 6 Zeichen signifikant. Unterschiede zwischen Klein- und Großbuchstaben sind implementierungsabhängig.
- interne Namen haben nur **innerhalb einer Source–Datei** Gültigkeit. Die ersten 32 Zeichen sind signifikant.

Beispiele:

- **Gültige Namen**:

  I, a5, _Bezeichner_1, this_is_a_very_long_long_name

- **Ungültige Namen**:

  do, 1000-mips, Zähler, A$%B

Empfohlene Regeln:

- Für **Makronamen**, insbesondere numerische Werte, nur **Großbuchstaben** verwenden.
- Der Unterstrich “_” als erstes Zeichen ist für (interne) Bibliotheksfunktionen reserviert.

Konstanten

Konstanten bezeichnen **feste Werte** eines bestimmten Typs:

- Integer-Konstanten vom Typ
- Gleitkommakonstanten vom Typ oder
- Character-Konstanten vom Typ
- Zeichenfeld-Konstanten vom Typ Feld von
- Aufzählungskonstanten vom Typ

Der Konstanten-**Typ ergibt sich aus** dem **angegebenen Wert**.

Integer-Konstanten

- **Dezimale** Angabe: Ziffern von bis ; falls mehrziffrig, muss die erste Ziffer 0 verschieden sein, z.B.: **15, 0, 32**
- **Oktale** Angabe, mindestens 2 Ziffern, stets mit der Ziffer 0 eingeleitet: **013**, entspricht dezimal den Zahlen 5 und 11.
- **Hexadezimale** Angabe, werden mit “x” eingeleitet, danach die Ziffern 0 bis 9 sowie die Buchstaben a bis f (oder A - F): **xff** bzw. **xa** entsprechen 255 bzw. 10.

Ein angehängtes “u” bzw. “U” definiert eine vorzeichenlose Konstante, “l” bzw. “L” eine lange Integer-Zahl: **255u, 1000L, 2047ul**.

Typ-Zuordnung

Konstante	Typ
Dezimalzahl ohne Suffix	int, long int, unsigned long int
Oktal- oder Hexadezimalzahl ohne Suffix	int, unsigned int, long int, unsigned long int
Suffix u oder U	unsigned int, unsigned long int
Suffix l oder L	long int, unsigned long int
Suffix u/U und l/L	unsigned long int

Beispiel: Seien Länge int = 16 Bits und Länge long = 32 Bits:

int	(Wertebereich: [-2^{15},^{15}-] ≙ [-32768,32767])
32000u	unsigned int (Wertebereich [0,2^{16}-])
33000	long int (Umwandlung nach long, da als int nicht darstellbar)

Gleitkomma-Konstanten

Darstellung wie auch bei anderen Sprachen gebräuchlich:

- Dezimalzahl mit Punkt: .14
- Ganzzahl mit Exponent: e-02
- Mischung aus beiden: .314E1

Standardmäßig sind Gleitkommakonstanten vom Format double. Anhängen eines f oder F deklariert sie als float, durch Anhängen von l oder L als double.

- .14f
- e-02l

Anmerkung: **Konstanten** werden vom Compiler ausgewertet, auch **Konstanten-Ausdrücke**, und nicht erst zur Laufzeit.

Character-Konstanten

Character-Konstanten sind Zeichen des verwendeten Zeichen-satzes, in einfache Hochkommas eingeschlossen. Hochkommas selbst sind mit Backslash zu escapen, ein Backslash ebenfalls sowie weitere Spezialzeichen:

’a’	Der Buchstabe a
’\\’	Der Backslash
’\’’	Einfaches Hochkomma
’\n’	NewLine als Spezialzeichen

Weitere Spezialzeichen:

'\a'	Bell	'\b'	Backspace	'\f'	Formfeed	'\r'	Carriage Return
'\t'	Htab	'\v'	Vtab	'\000'	Oktaldarstellung	'\xhh'	Hex-Darstellung

String-Konstanten

String–Konstanten sind **Zeichenfolgen**, die in **Anführungszeichen** eingeschlossen sind. Escape-Sequenzen sind genau wie bei Character-Konstanten möglich. Der **Typ** einer String-Konstanten ist ein **Feld** von Einzelzeichen (jeweils der Länge 1 Byte)..

Bei String–Konstanten, die länger als eine Zeile sind, sind Teilzeilen mit "\" abzuschließen. Für String–Konstanten der Länge n legt der Compiler einen Speicherbereich der Länge n+1 an; an der (n+1)-ten Stelle wird \0 abgelegt.

```
“Dies ist eine Zeichfolge mit \t Escape-Sequenzen”
“Dies ist eine sehr, sehr, sehr, sehr, sehr, sehr, sehr lange\
Zeichfolge”
```

“123” steht fortlaufend im Speicher als: ’1’ ’2’ ’3’ ’\0’

Aufzählungs-Konstanten

Der **Aufzählungstyp** repräsentiert eine **Liste von benannten Integer-Konstanten** und wird durch das Schlüsselwort enum definiert. Die einzelnen Integerwerte **beginnen** normalerweise **bei 0** und ergeben sich durch die Reihenfolge in der Deklaration:

```
enum { mo, di, mi, do, fr, sa, so };
```

In dieser Deklaration hat z.B. mi den Wert 2.

3. Der Präprozessor

Aufgaben des Präprozessors:

- Einfügen zusätzlicher Quelldateien und Bibliotheken(Header-Files)
- Bedingte Übersetzungen
- Makrodefinitionen und Makroersetzungen

Der Präprozessor wird über **Direktiven** gesteuert. Eine Direktive wird **mit '#' eingeleitet**. Ist sie länger als eine Zeile, kann sie nach einem '\' am Zeilenende **fortgesetzt** werden.

Aktionen:

- Ersetzen von **Trigraph-Sequenzen**
- Löschen der Kombination **Backslash-Newline**
- Aufspalten des Quellcodes in **Grundelemente**
- Entfernen aller **C-Kommentare**
- Behandeln der **Direktiven einschließlich Makroersetzung**

Einfügen von Dateien

Dateien werden durch die Direktive

#include <*dateiname*> oder
#include "<*dateiname*>"

eingefügt, genauer: es wird auf die einzufügende Datei (auch geschachtelt) umgeschaltet. Bei Angabe eines **kompletten Pfadnamens** wird die entsprechende Datei includiert. Sind relative Dateinamen angegeben, so erfolgt ein Suchvorgang in **implementierungsabhängigen Directories**, z.B. **/usr/include**. Der Suchpfad kann im Compile-Kommando erweitert werden (Option -I).

So inkludiert z.B. die Präprozessoranweisung **#include <stdio.h>** unter UNIX/LINUX die Datei **/usr/include/stdio.h** für Standard-Ein/Ausgabe. In der Form **#include "dateiname"** wird zuerst in der **lokalen Umgebung** (z. B. im aktuellen Verzeichnis) gesucht. Die **Dateinamen** sind auch **über Makrodefinitionen** angebbar.

Makros

Mit Makros kann man **häufig verwendete Texte** abkürzen oder **parametrisieren**. Nach der Definition eines Makros wird bei seinem Auftreten der Makro durch den Makro-Rumpf (auch Ersatztext genannt) ersetzt, falls er **nicht innerhalb** einer **String**- oder **Character**-Konstanten auftritt:

1. ```
   #define MAKRO        \
                        \
   int i = 10;       \
   return i*i;
   ```

2. ```
   #define PI 3.14159
   printf(”PI = %f\n”,PI);
   ```

   generiert:

   ```
   printf(“PI = %f\n“,3.14159);
   ```

Allgemein:

1. #define makroname ersatztext
2. #define makroname(par$_1$,par$_2$,...) ersatztext

Im zweiten Fall werden beim Aufruf von **makroname** die einzelnen Parameter ersetzt.

Beispiel:

```
#define BIG_CONST 10
#define SM_CONST(b) (BIG_CONST+(b))
int a[BIG_CONST], b[SM_CONST(6)];
```

Ergibt:

```
int a[10], b[(16)];
```

Anmerkungen

Der **Ersatztext kann auch leer sein**; in diesem Fall dient ein Makroaufruf oft zum Setzen eines Flags, das sich mit #if defined abfragen lässt.

Schritte:

1. Sind Makroparameter definiert, darf zwischen Makroname und '(' kein Blank stehen;
2. Zeilenenden werden durch "\" als letztes Zeichen einer Zeile aufgehoben;
3. Parameter werden identifiziert durch Kommas innerhalb des obersten Klammernpaares;
4. innerhalb der aktuellen Parameter werden Makroaufrufe ersetzt;
5. Ersetzen der formalen durch die aktuellen Parameter;
6. Ersetzen des Makroaufrufes durch den Makrotext.

Vorsicht bei **aktuellen Parametern**, die aus **Ausdrücken** bestehen: Problem der Gewichtung von Operatoren. Ausweg: **korrekte Klammernsetzung**:

```
#define MULT(a,b) a*b
```

Dann liefert

MULT(x+y,u+w) die Ersetzung x+y*u+w

und nicht, wie erwartet,

(x+y)*(u+w).

Korrekte Makrodefinition wäre z.B.:

```
#define MULT(a,b) ((a)*(b))
```

Makroaufrufe **werden entfernt** (undefiniert!) durch

```
#undef <makroname>
```

Bedingte Makroersetzung

Durch Direktiven ist es möglich, bestimmte Teile einer Quelldatei zu übersetzen oder nicht:

```
#if test_1
    Codestück_1
#elif test_2
    Codestück_2
#else
    Codestück_3
#endif
```

Regeln: Jede #if-Direktive endet mit genau einem #endif, dazwischen sind `#elif`'s möglich. test_*i* ist ein **arithmetischer Ausdruck**, der als **False** oder **True** bewertet wird, jenachdem, ob er 0 oder ≠0 ist.

Für test_i ist auch die **Direktive defined** erlaubt, mit der sich abfragen lässt, ob ein Makro definiert ist:

```
#define UNIX            /* Makro "UNIX" wird definiert */
char *sys =
#if defined(VMS) /***** Anfang Makroroabfrage *****/
    "vms"
#elif defined (DOS)
    "DOS"
#elif defined (UNIX)
    "UNIX"
#endif                  /***** Ende Makroabfrage *****/
    ;
```

Präprozessor erzeugt daraus:

```
char *sys =
    "UNIX"
    ;
```

Weitere Direktiven

#pragma **_Compilerpragma_**
Übermittlung von Informationen an den Compiler, implementierungsabhängig

Sonst nur Whitespace: Ohne Wirkung

#line Zeilennummer "Datei"
Änderung der internen Zeilennummer und des Dateinamens. Diese Variablen sind über __LINE__ und __FILE__ verfügbar.

#error token Compiler gibt Fehlermeldung mit *token* als Text aus.

Makro-Operatoren

defined(Makro) — 0 oder 1, jenachdem ob *Makro* definiert ist oder nicht

— NächstesArgument wird in "..." eingeschlossen

```
#define a(dir)  #dir
a(/usr/local) erzeugt /usr/local
```

— Verkettungsoperator : Die beiden Argumente rechts und links neben ## werden zu **einem** String verkettet:

```
#define m(a,b) a ## b
m(u,_v) generiert u_v
```

Vordefinierte Makros	
__LINE__	Dezimalkonstante mit momentaner Zeilennummer
__FILE__	Zeichenfolge, die den Namen der momentanen Datei enthält
__DATE__	Aktuelles Datum in der Form **"Dec 13 1999"** als Stringkonstante
__TIME__	Aktuelle Uhrzeit in der Form **“14:42:44”** als Stringkonstante
__STDC__	**1**, wenn der aktuelle Compiler ein ANSI-C-Compiler ist

4. Typen

Jede in C verwendete Größe wie Variablen, Konstanten oder Funktionen **besitzt einen Typ**. Der Typ einer Konstanten ergibt sich aus deren Wert, die Typen von Variablen und Funktionen aus Definition oder Deklaration. Da der Compiler Typunverträglichkeiten erkennen kann, werden dadurch Fehler verringert.

Elementare **Basistypen**:

- Integer-Typen: int, enum, char
- Gleitkomma-Typen: float, double, long
- Leerer Typ: void

Gruppierte **Datentypen** wie z.B. Felder oder Strukturen sind daraus **abgeleitet**.

Integer-Typen

Mittels Integer-Typen lassen sich folgende Werte darstellen:

- Ganzzahlige Werte mit und ohne Vorzeichen
- Bit-Vektoren und Bit-Felder innerhalb von Strukturen/Unions
- Darstellung für False (0) und True (≠0).
  Boolsche Werte im engeren Sinne existieren in C nicht, erst in C++
- Character-Werte als Integer-Zahlen (Interncode, ASCII oder EBCDIC)

Varianten von Integer-Typen: short/long, mit/ohne Vorzeichen.

Vorzeichenbehaftete Integer-Typen (signed)

- **short** (oder **short int**),
- **int**,
- **long** (oder **long int**)

Es gilt:

$$\textbf{Länge}_{short} \leq \textbf{Länge}_{int} \leq \textbf{Länge}_{long}.$$

Die Voreinstellung signed kann entfallen.

Beispiele:

```
int i;            /* Integer-Zahl mit Vorzeichen */
short j;          /* kurze Integer-Zahl mit Vorzeichen */
long int k;       /* lange Integer-Zahl mit Vorzeichen */
signed long l;    /* lange Integer-Zahl mit Vorzeichen */
```

Die Länge von int ist i. A. gleich der Rechner-Wortlänge.

Vorzeichenlose Integer-Typen (unsigned)

- **unsigned short** (oder **short int**),
- **unsigned int** (oder **unsigned**),
- **unsigned long** (oder **unsigned long int**)

Es gilt: Länge(unsigned short) ≤ Länge(unsigned int) ≤ Länge(unsigned long).

unsigned **muss** angegeben werden, falls erwünscht.

Beispiele:

```
unsigned int i;          /* Integer-Zahl ohne Vorzeichen */
unsigned j;              /* Integer-Zahl ohne Vorzeichen */
unsigned short jj;       /* kurze Integer-Zahl ohne Vorzeichen */
unsighed long int k;     /* lange Integer-Zahl ohne Vorzeichen */
```

Von **int** unterscheiden sich **unsigned int** Zahlen i.A. durch einen doppelt so großen positiven Wertebereich. Sie belegen den gleichen Speicherplatz wie vorzeichenbehaftete Integer-Zahlen, besitzen aber relevante **Vorzeichenbits**.

Character-Typen (char)

- **char**
- **signed char**
- **unsigned char**

Die Angabe von char allein **sagt nichts über das Vorzeichen aus**. Dies ist implementierungsabhängig. Die Länge von char ist i.A 1 Byte (8 Bit). char-Werte sind Integer-Werte und somit in den entsprechenden Ausdrücken verwendbar.

```
char c;                 /* character */
unsigned char d;        /* character ohne Vorzeichen */
signed char e;          /* character mit Vorzeichen */
```

Aufzählungs-Typ (enum)

Mittels Aufzählungstypen ist es möglich, **Konstanten Integer-Werte zuzuweisen** und diese über einen **Namen** anzusprechen:

enum bezeichner { list of constants };

bezeichner ist ein Name für den Aufzählungstyp, und in *list* werden die einzelnen Bezeichner bzw. an beliebiger Stelle "Bezeichner = Konstante"-Paare aufgeführt, durch Komma getrennt. Defaultmäßig beginnen die Konstanten bei 0.

bezeichner kann als quasi neuer Typ verwendet werden.

Beispiel:

```
enum MON {jan=1,feb,mar,apr,may,jun,jul,aug,sep,oct,nov,dec};

enum MON  monat;
          monat = 0;          /* Fehler, nicht erlaubt */
          monat = may;   /* OK */
```

Gleitkomma-Typen (float)

Gleitkomma-Typen dienen der Darstellung von reellen Zahlen. Man unterscheidet

- float
- double
- long double

Der Typ float bezeichnet einfach genaue Gleitkommazahlen, double mindestens einfach genaue Gleitkommazahlen und long double solche, die mindestens so groß sind wie double, d.h.

$$\text{Länge}_{\text{float}} \leq \text{Länge}_{\text{double}} \leq \text{Länge}_{\text{long double}}$$

double bedeutet nicht doppelte Genauigkeit (wie z.B. in Fortran). Der Compiler wandelt alle Gleitkommakonstanten (soweit ohne suffix) in double.

Beispiel:

```
float x;
double dx;
long double ldx;
```

Leerer Typ (void)

Der **void**-Typ hat keinen Wert, und es gibt keine zulässigen Operationen. Er wird in folgenden Zusammenhängen verwendet:

- **Als Generischer Zeiger: void *** als Funktions-Rückgabewert, der in Zeigerwerte eines bestimmten Typs umgewandelt werden muss.

  Beispiel: malloc liefert **void *** als Rückgabetyp.

- Definition von **Funktionen ohne Rückgabewert**. Dies ist genau der **Unterschied** zwischen **Prozeduren** und **Funktionen**.

- **void *** als Funktions-Parameter: als Parameter darf eine beliebiger Zeiger-Wert angegeben werden.

- **void** als einziger Funktions-Parameter in der Deklaration heißt: Die Funktion/Prozedur hat **keine Aufruf-Parameter**.

void-Beispiel:

```
/* Definitionen */

            void ptr;               /* Generischer Zeiger */
            void f1();              /* Die Funktion f1 liefert kein Ergebnis */
            int f2(void);           /* Die Funktion hat f2 keinen Parameter */

/* Anweisungen */

            (void) getchar();       /* Ignoriere das Funktionsergebnis */
```

Zusammengesetzte Typen

Zusammengesetzte Typen werden **induktiv über Basistypen** T_1, T_2,... definiert:

- Zeiger auf T_1;
- Feld von T_1;
- Struktur von T_1,T_2,...;
- Funktion, die T_1 liefert.

Zeiger-Typen

Zu jedem bislang definierten Typ lässt sich der jeweilige **Zeiger-Typ** erzeugen. Er kann als die **Adresse eines Objekts** eines bestimmten Typs interpretiert werden. Bei der Deklaration werden Zeiger-Variable durch einen einleitenden * gekennzeichnet.

Für alle Zeigertypen gemeinsam existiert der Null-Zeiger (Konstante bzw. Präprozessor-Konstante), definiert in **<stddef.h>**. Seine Verwendung ist applikationsabhängig.

Operatoren:

- Der **Adress-Operator &** liefert die Adresse eine Objektes.
- Der **Dereferenzierungs-Operator *** liefert zu einem Zeiger den Wert, auf den der Zeiger zeigt.

Es gilt stets: ***(&p)** ist **identisch** mit **p**.

Beispiele:

```
int *p;        /* Zeiger auf int */
int q,r;       /* vom Typ int */
p = &q;        /* Adresse von q */
r = *p;        /* Inhalt von q an r zuweisen */
               /* wie r = q; */
```

Feld-Typen

Zu jedem bislang definierten Typ lässt sich der jeweilige **Feld-Typ** erzeugen (auch Vektor oder Array genannt). Dabei handelt es sich um eine homogene Ansammlung von Daten des gleichen Typs, die sequenziell im Speicher abgelegt werden.

Die einzelnen Daten werden über Indizes angesprochen.

```
int vec[10];        /* vec ist Feld von 10 int Variablen */
char *p[9];         /* p ist Feld von 9 Zeigern auf char */
vec[0] = 11;
```

Der **erste Feld-Index** hat stets den Wert **0**, der **letzte** den Werte **n-1**, falls **n** die Feldlänge ist.

Wichtig:

Zwischen **Feld** und **Zeiger** besteht folgender Zusammenhang: Der Feldname kann als Zeiger verstanden werden, der auf das erste Feldelement zeigt.

Das heißt im obigen Beispiel: **vec** hat numerisch den gleichen Wert wie **&vec[0]**, und **&vec[i]** hat den gleichen Wert wie **vec + i*elementlänge** (für $0 \leq i \leq feldlänge$).

Strings (Zeichenketten):

Sie lassen sich auch als **Feld von char** erklären:

```
char s[4];
s[0] = ’a’;
s[1] = ’b’;
s[2] = ’c’;
s[3] = ’\0‘;
```

Beachte: **letztes Zeichen** eines Strings (einer Zeichenkette) muss das **NULL-Zeichen** sein! Bei seiner **Länge** zählt das **NULL-Zeichen nicht** mit.

Mehrdimensionale Felder lassen sich entsprechend aufbauen:

```
a[10][20];
a[3][4] = 2;
```

Die Interpretation ist folgende: **a[0] .. a[9]** könnten (!!) auch als **int**-**Felder** jeweils der Länge 20 angesehen werden.

!!Wichtig!!: Beachte den **Unterschied** zu:

```
<typ> *a1[10];
```

Während des Programmlaufes lässt sich mit **a1** genauso arbeiten wie mit **a**, aber: bei **a** wird Speicher für 200 Integer-Zahlen reserviert, bei **a1** lediglich Speicher für 10 Pointer des Typs <typ>, nicht aber für die einzelnen Felder. **Vorteil bei a1**: Die einzelnen Felder können unterschiedlich lang sein. **Aber**: Sie sind **explizit zu allokieren**!

Feldgrößen-Angaben:

- Explizit innerhalb des Klammernpaares [...]
- Durch Angabe der Feld–Initialwerte:

```
int a[] = {1,2,3}; /* int–Feld der Länge 3 */
```

- Weglassen der Größenangabe bei Feld-Parametern innerhalb eines Funktions aufrufes (hier aber nur bei der Angabe der 1. Dimension), bei externen Definionen und Vorwärts–Deklarationen.

  Beispiele:

```
func0(...,int a[][10][20],...);
func1(...,int a1[][][],...); /* Fehler! */
extern char c[];
char *v[];
char *v1[20];
```

Struktur-Typen

Strukturen dienen der Zusammenfassung **unterschiedlicher Daten** unter **einem** Namen. Die einzelnen **Daten** nennt man **Komponenten**.

C kennt drei Struktur–Typen:

- **struct**
- **union**
- Bitfelder innerhalb von **struct**-Typen

Ein **struct**-Typ wird folgendermaßen definiert:

struct *bezeichner* { *Komponenten...* };

Der optionale *bezeichner* (oder *tag*) benennt den Strukturtyp. Die Komponenten besitzen Typen wie bislang diskutiert, zusätzlich auch Struktur-Typen und werden **hintereinander im Speicher abgelegt**. Merke: Strukturen gleicher *bezeichner* besitzen gleiche Größe!

Beispiel:

```
struct a_type
{
        int a;
        double b;
        int *f(int);
};

struct a_type bn_s;  /* bn_s ist Struktur mit a_type-Layout */
```

Möglich wäre auch:

```
struct a_type
{
        int a;
        double b;
        int *f(int);
} bn_s;         /* bn_s ist Struktur mit a_type-Layout */
```

Zugriff auf die einzelnen Komponenten:

strukturname.komponentenname,

d.h. im obigen Beispiel:

```
int i = bn_s.a;
double x = bn_s.b;
int *j = bn_s.f(1);
```

Zeiger auf Strukturen:

Zu Struktur-Typen lassen sich auch Zeiger bilden:

```
struct a_type
{
        int a;
        double b;
        int *f(int);
} *bn_s;    /* bn_s ist Zeiger auf Struktur mit a_type-Layout */
```

Zugriff auf die Elemente mit Pfeil-Operator (oder auch mit *–Operator):

struktur_zeiger–>komponente, also

```
...
bn_s->a = 1;   /* oder: (*bn_s).a = 1; */
double x = (*bn_s).b;
bn_s–>f(1);
```

Verschachtelung von Strukturen:

Innerhalb von Strukturen dürfen Strukturen auftreten (weil **alle** Typen in Strukturen auftreten dürfen!), sogar Zeiger auf Strukturen vom identischen Layout:

```
struct  complex
{
    double re;
    double im;
}

struct s_struct
{
    struct complex c;
    struct s_struct *s1;
    /* Zeigt auf eine Struktur gleichen  Bezeichners (s_struct) */
}
```

Bitfelder

Innerhalb von Strukturen dürfen **Bitfelder** definiert werden, die durch **int**-Typen repräsentiert werden (**signed** oder **unsigned**). Sie dürfen benannt oder unbenannt sein.

Sie unterscheiden sich von normalen **int**-Typen durch eine Längenangabe in Bits:

```
struct bb_type
{
        int ba:7;   /* Bitfeld, 7 Bits lang */
        int :25;    /* Unbenanntes Bitfeld, 25 Bits lang */
        int :0;     /* unbenannt, dient als int-Alignment */
}
```

Vorsicht: Strukturen mit Bitfeldern haben **kein portables Layout**! Sie werden in erster Linie bei der maschinennahen Programmierung verwendet.

Vereinigungs-Typ (union)

Vereinigungs-Typen, durch das Schlüsselwort definiert, unterscheiden sich von Strukturen nur dadurch, dass die Komponenten **nicht hintereinander** im Speicher abgelegt werden, sondern **alle auf der selben Speicherstelle**. Die Länge einer **union** ist die maximale Komponenten-Länge.

```
union u_t   {
                int a;
                char f[4];
                struct
                {
                    int a1:1;
                    ...
                    int a32:1;
                } s;
            } u;
```

Mittels u.f[...] sich die **einzelnen Bytes** von a **ansprechen,** mit u.s.a1,..., u.s.a32 die einzelnen Bits.

Funktions-Typen

Aus einem Typ **T** lassen sich Funktionen bilden, die einen Rückgabewert vom Typ **T** liefern. Dies kann in zweierlei Zusammenhang auftreten:

1. **Definition** einer Funktion (mit Rumpf!):

```
int square(int a)
{
        return a*a;
}
```

2. **Deklaration** einer Funktion (ohne Rumpf!):

```
int sqare(int);
```

Erscheint ein **Bezeichner** einer solchen Funktion nicht in Zusammenhang mit einem Funktionsaufruf, so wird er **automatisch konvertiert in einen Zeiger** auf eine Funktion, die den entsprechenden Typ liefert.

Das heißt: Funktionen (bzw. deren Namen) können auf 3 Arten referiert werden:

- Als Funktionsaufruf:

```
f()
```

- Funktion als Aktualparameter eines anderen Funktionsaufrufes

```
g(f,1,2)
```

- Zuweisung an einen passenden Zeigertyp (s.o.):

```
int (*g)(int);
g = f;
```

Typ-Namen

Die **typedef**-Deklaration erlaubt die Definition neuer Typen ***ntyp*** aus bereits vorhandenen *typ*:

typedef *typ ntyp;*

ntyp kann ab Deklaration als neuer Datentyp verwendet werden:

```
typedef unsigned int uint;
typedef struct
{
        double re;
        double im;
} complex;
complex u,v,w;          /* Komplexe Zahlen u,v,w (Strukturen) */
typedef (*func_type)(int, complex *,long double);
                        /* func_type = Funktionstyp */
func_type sig[10];      /* Feld von 10 Funktionen des Typs func_type */
```

Typ-Qualifizierer (const**,** volatile**)**

ANSI-C kennt zwei Typ-Qualifizierer:

1. **const**

   const-Objekte dürfen nach der Initialisierung nicht mehr das explizites oder implizites Ziel von Zuweisungen sein. Somit lassen sich konstante Objekte erzeugen.

```
const double PI = 3.1415927;
...
PI = 1.11;              /* Fehler! */
```

2. **volatile**

   Implementierungsabhängig. Üblicherweise verwendet man das Attribut **volatile** für Variablen, die durch Ereigisse außerhalb des Programmes verändert werden können, z.B. durch Interrupts.

5. Deklarationen und Definitionen

Festlegung der Bedeutung von Bezeichnern, d.h.

- Speicherklasse
- Typ
- Bezeichner selbst

Die **Speicherklasse** legt ***Gültigkeitbereich*** und ***Lebensdauer*** fest.

Der **Typ** bestimmt den ***Datentyp des Bezeichners***.

Syntax:

[speicherklasse] [typangabe] bezeichner [,bezeichner ...] ;

Der Bezeichner kann zusätzliche Angaben enthalten wie * davor als Zeiger, [] dahinter zur Feldangabe oder auch () dahinter als Funktionsangabe usf.

Durch eine **Deklaration** wird im allgemeinen **kein Speicherplatz reserviert**.

Reservierung von Speicher durch eine Erweiterung der Deklaration zur **Definition**, aber:

Unterschiede nur in wenigen Fällen!

Beispiele für Deklarationen:

```
int x;
extern const int y;
int *i1_zeiger, i2;
double d[17], *d[12][13];

int *f(int x, double y);
double (*g[10])(void (*vf)(void)) ;
```

Wo sind Deklarationen erlaubt?

- Außerhalb von Funktionen als **externe** oder **globale Deklarationen**
- Am Beginn eines Blocks, also auch am Beginn eines Funktionsrumpfes
- In einem Funktionskopf (d.h. innerhalb der runden Klammern)

Beispiel:

```
/* Datei test.c */
int x;                          /* extern */
static int y;                   /* source global */
void fu(int *,...);             /* extern */
int main(int argc, char **argv) /* definition */
{
        int x1,x2,x3;           /* lokal */
        fu(&x1,&x2,&x3,NULL);
lab:                            /* Label */
        return 0;
}
```

Gültigkeit deklarierter Bezeichner:

Extern:	Ab Deklarationspunkt bis Dateiende
Funktionsparameter:	Im Funktionsrumpf
Lokal im Block:	Ab Deklarationspunkt bis Blockende, auch bei Speicherklasse **`extern`**
Labels:	Gesamte Funktion

Mehrfachbenutzung von Bezeichnern **möglich**, aber nicht empfohlen:

```
struct xyz { char *a,*xyz; }
int xyz;
```

Sichtbarkeitsregeln bei Namenskonflikten:

- Deklaration **formaler Funktionsparameter** überschreibt **globale Deklarationen.**
- **Block-lokale** Deklaration überschreibt Deklaration **außerhalb** des Blocks.

Beispiel:

```
int pi;
double f(char *pi)
{
    if (pi == NULL)
    {
        double pi = 3.14159;
        return pi;
    }
}
```

Lebensdauer deklarierter Objekte:

- **Statisch:**

  für die gesamte Programmlaufzeit:

  - Alle Funktionen
  - Alle als **extern** deklarierten Variablen
  - Alle als **static** deklarierten Variablen

- **Automatisch**:

  Freigabe bei Verlassen des zugehörigen Blocks

- **Dynamisch**:

  Bei Verwendung bestimmter C-Laufzeitfunktionen, aber kein Bestandteil des C-Sprachumfanges (malloc, free)

Bindung von Bezeichnern:

In welchen Source–Dateien eines Programmes bezieht sich ein Bezeichner gleichen Namens auf das gleiche Objekt?

- **Externe** Bindung: in allen
- **Interne** Bindung, dateiglobal: innerhalb dieser Datei
- **Ohne** Bindung: einmalig, z.B. innerhalb eines Blockes

Speicherklassen:

Die Speicherklasse beeinflusst bei einem Objekt dessen **Bindung**, **Lebens-dauer** und **Gültigkeitsbereich**.

auto:

am Blockbeginn erlaubt. Objekterzeugung bei Eintritt in den Block, Objektfreigabe bei Verlassen des Blocks. Gültigkeit nur innerhalb des Blocks.

register:

für den Zugriff auf häufig benutzte Variable.Empfehlung an den Compiler, sie in Registern abzulegen. Keine Garantie! Der Adress-Operator **&** ist nicht erlaubt! Zulässig für Block-lokale Variable sowie Funktionsparameter.

static:

für Funktionen und Variable. Lebensdauer statisch. Bei Datei-globalen Variablen Gültigkeit auf Datei begrenzt, bei lokalen Variablen auf den sie umgebenden Block. Initialisierung nur einmal, durch den Linker.

extern:

erlaubt bei externen Deklarationen und zu Beginn eines Blocks. Statisch, **global im Programm gültig**.

Beispiel:

```
extern int lock;                    /* global */
register int i;                     /* Datei-lokal */
static int f(register int i);       /* Datei-lokal */

auto int j;                         /* Error,nicht erlaubt */
static int k;                       /* lokal, behält Wert */

extern int g(void);                 /* global, extern */
static int f(register int i)
{
        static int j;               /* lokal */
        extern int g(void);         /* global */
        ...
}
```

Typangaben, Zusammenfassung:

Als **Typangaben** sind folgende Schlüsselworte erlaubt:

- **void**
- Alle numerischen Typen wie **short, int, long, float, double,** mit Zusatzattributen wie
  - **struct**
  - **union**
  - **enum**
- **typedef**

Typqualifizierer sind erlaubt:

- **const** /* Dem Objekt darf nichts zugewiesen werden */
- **volatile** /* Implementierungsabhängig */

Spezifikatoren für Bezeichner bei Deklarationen:

1. ***** *vor* Bezeichner: Zeigertyp

```
*p;
```

2. **[ ]** *hinter* Bezeichner: Feld-Typ, mit Längenangaben, die erste Längenangabe kann entfallen

```
*a[][30], b[10];
```

3. Funktionen: Auf den Namen folgen, in runde Klammern eingeschlossen, die Deklaratoren für die Parameter; die Angabe von Bezeichnern ist nicht erforderlich. Als letzter Parameter ist die Eklipse **...** erlaubt.

```
f(double l, struct _st *h_struct,...);
*f(double, struct _st *,...);
```

Bedeutung der Eklipse als Parameter-Deklarator:

Nach dem letzten deklarierten Parameter können beim Funktionsaufruf noch beliebig viele weitere Parameter des gleichen letzten Typs folgen. Für geeignete Abbruchkriterien ist der Programmierer selbst verantwortlich. Hilfen zur Auswertung solcher Parameterlisten bietet

```
#include <stdarg.h>
```

Beispiel: E/A-Funktionen wie **`printf`** oder **`scanf`**.

Bei **externen** Funktionen wird der Deklarator auch *Funktions-Prototyp* genannt.

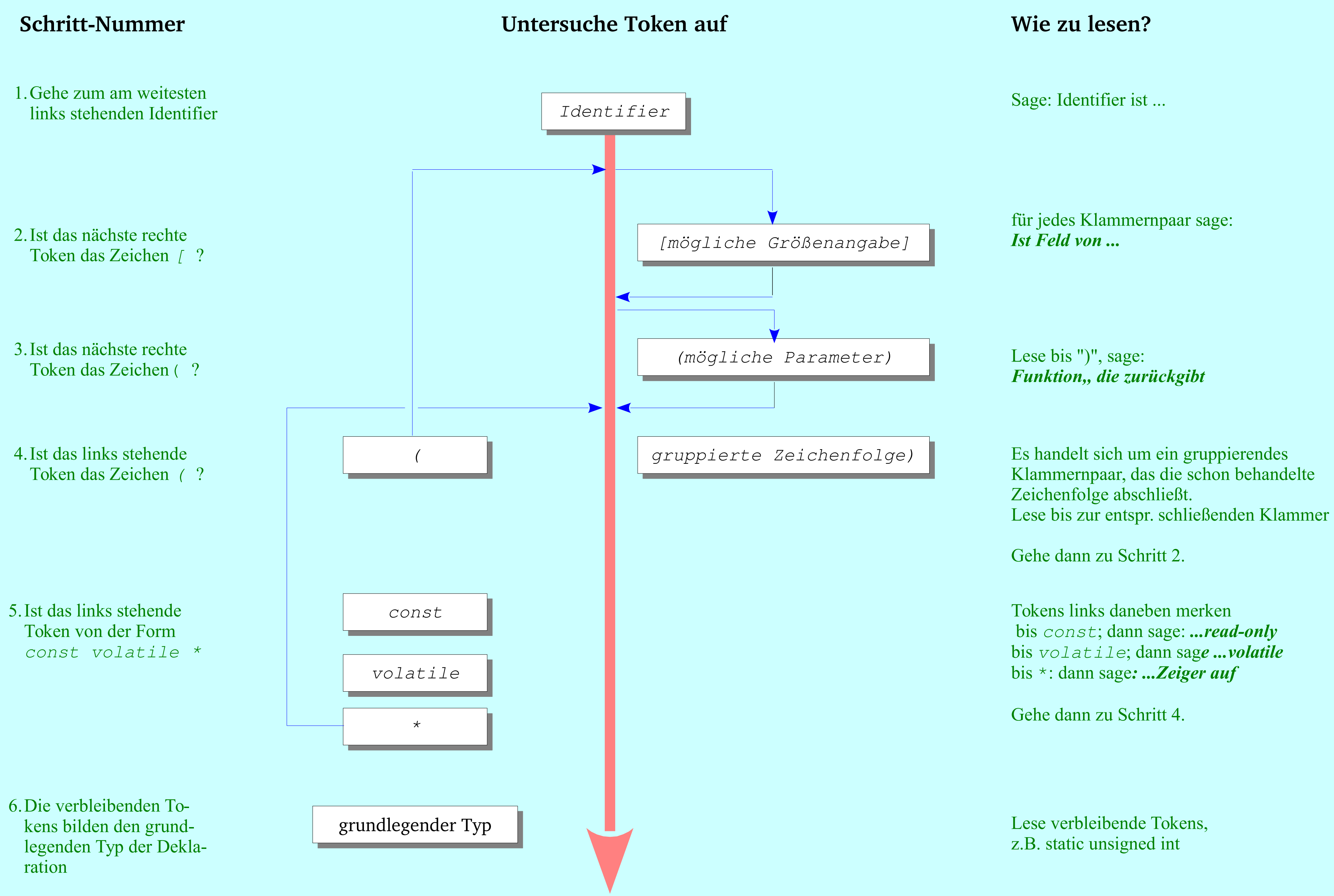
Die alte Form der Deklaration (leerer Parameter-Deklarator) sollte **nicht verwendet** werden! Desgleichen keine implizite Deklaration (implizite Deklaration beim ersten Aufruf)!

Deklarator-Prioritäten:

Vorrangregeln (abnehmend von 1 nach 3):

1. Klammerung
2. Funktionsdeklaratoren, Feld-Deklaratoren
3. Zeiger-Deklaratoren

Die Verarbeitung durch den C-Compiler erfolgt mittels rekursivem Abstieg unter Verwendung der Vorrangregeln:


Schritt-Nummer
Untersuche Token auf
Wie zu lesen?
1. Gehe zum am weitesten links stehenden Identifier
Identifier
Sage: Identifier ist ...
2. Ist das nächste rechte Token das Zeichen [?
[mögliche Größenangabe]
für jedes Klammernpaar sage:
Ist Feld von ...
3. Ist das nächste rechte Token das Zeichen (?
(mögliche Parameter)
Lese bis ")", sage:
Funktion,, die zurückgibt
4. Ist das links stehende Token das Zeichen (?
(
gruppierte Zeichenfolge)
Es handelt sich um ein gruppierendes Klammernpaar, das die schon behandelte Zeichenfolge abschließt.
Lese bis zur entspr. schließenden Klammer
Gehe dann zu Schritt 2.
5. Ist das links stehende Token von der Form const volatile *
const
volatile
*
Tokens links daneben merken
bis const; dann sage: ...read-only
bis volatile; dann sage ...volatile
bis *: dann sage: ...Zeiger auf
Gehe dann zu Schritt 4.
6. Die verbleibenden Tokens bilden den grundlegenden Typ der Deklaration
grundlegender Typ
Lese verbleibende Tokens,
z.B. static unsigned int

Beispiel: Die `signal`-Funktion:

```
(* (*signal)(int sig,void (*FUNC)(int)) ) (int);
```

Besser:

```
typedef void (*SIG_FUNC)(int);
SIG_FUNC *signal(int ,SIG_FUNC);
```

Übung: Was bedeutet

```
char *(**c[10][20]) (int **[], double (*)(void));
char *const (*next)(int,double);
```

Unterschied zwischen Deklarationen und Definitionen:

relevant bei externen Variablen und Funktionen!

Variable:

Durch die Deklaration wird kein Speicherplatz belegt. Definition einer externen Variablen durch Initialisierung in genau einer der Dateien, aus denen das Programm besteht!

a1.c: int *a;
a2.c: int *a = NULL;

Funktionen:

Ist das erste relevante Token hinter der schließenden Klammer kein Semikolon, sondern "**{**", wird damit eine Funktionsdefinition eingeleitet!

Beispiel:

```
void f(int);                              /* deklarator von f, prototytp */
main(int argc, char **argv)
{
    for (;argc>=0;argc--) f(argc+10);
    return 0;
}
void f(int i)                             /* definition von f */
{
    printf("%d\n",i);
}
```

Initialisierung von Variablen:

Formal:

Deklarator = Ausdruck;

Eine solchermaßen deklarierte Variable wird damit zum einen auch definiert, zum anderen wird sie beim Anlegen mit dem Ausdruck initialisiert.

Folge:

Externe und statische lokale Variable werden nur einmal beim Programm-start initialisiert, automatische jedesmal beim Anlegen der Variablen. Desweiteren sind externe und statische lokale Variable nur mit konstanten Ausdrücken initialisierbar, automatische dagegen auch mit Variablen-Ausdrücken.

Als Variable sind neben einfachen Variablen auch Felder und Strukturen initialisierbar.

Statische arithmetische Variable werden, wenn nicht anders angegeben, mit 0 initialisiert, statische Zeigervariable mit dem NULL-Pointer.

Beispiel:

```
static int a=3;
void f(void)
{
        int         *b = &a,
                    c = *b**b;
        struct      {int x; double y;} s = {1,2.10};
        char*f[] = {"123","456"},
                    g[4] = "123";
        return;
}
```

6. Operatoren und Ausdrücke

Ausdrücke: eine Folge von **Operanden** und **Operatoren** mit dem Resultat, dass

- ein Wert berechnet wird oder
- der Typ eines Objektes festgelegt oder geändert wird (in casts) oder
- Seiteneffekte erzeugt werden.

Die Operanden selbst können wiederum Ausdrücke sein!

Jeder Ausdruck besitzt einen Wert.

<table>
<tr><th colspan="2">Klassen von Operatoren</th></tr>
<tr><td>Zugriffs-Operatoren</td><td>[ ]   ( )   .   -></td></tr>
<tr><td>Arithmetische Operatoren</td><td>+ - * / % ++ -- += -= *= /= %=</td></tr>
<tr><td>Logische Operatoren</td><td>&& || !</td></tr>
<tr><td>Bit-Operatoren</td><td>& | ^ ~ &= |= ^= ~=</td></tr>
<tr><td>Sonstige Operatoren</td><td>( ) sizeof , ?: =</td></tr>
</table>

Einteilung nach **Anzahl beteiligter Operanden** (1, 2 oder 3):

- *Unäre* Operatoren
- *Binäre* Operatoren
- *Ternäre* Operatoren

Bei der Bewertung von Ausdrücken besitzt jeder Operator eine feste Priorität, die sich teilweise an den geläufigen Rechenregeln orientiert (”Punktrechnung geht vor Strichrechnung”). Siehe dazu auch die folgende **Tabelle**!

<table>
<tr><td>()</td><td>[]</td><td>.</td><td>-></td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td>!</td><td>~</td><td>&$_1$ (Pointer)</td><td>*$_2$ (Dereferenzierung)</td><td>sizeof</td><td>(typecast)</td><td>++</td><td>unäres -</td><td>-</td><td>unäres +</td></tr>
<tr><td>+ (Addition)</td><td>/</td><td>%</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td>>></td><td><<</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td>></td><td>>0</td><td><</td><td><=</td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td>& (Arithmetisch)</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td>^</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td>&&</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td>||</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td>?:</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td>=</td><td>+=</td><td>-=</td><td>*=</td><td>%=</td><td>&=</td><td>^=</td><td>|=</td><td><<=</td><td>>>=</td></tr>
<tr><td>,</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
</table>

<table>
<tr><th colspan="2">Erläuterungen</th></tr>
<tr><td>-></td><td>Dereferenzieroperator bei Strukturen</td></tr>
<tr><td>&$_1$</td><td>Adressoperator</td></tr>
<tr><td>*$_2$</td><td>Dereferenzieroperator</td></tr>
<tr><td>/</td><td>Division (für <code>int</code> und <code>float</code>)</td></tr>
<tr><td>%</td><td>Modulo (Rest)</td></tr>
<tr><td>>>, <<</td><td>Bitshift</td></tr>
<tr><td>&, |</td><td>arithmetisches (bitweises) UND, ODER</td></tr>
<tr><td>&&, ||</td><td>logisches UND, ODER</td></tr>
<tr><td>^</td><td>Exlusives ODER</td></tr>
<tr><td>sizeof</td><td>Größe des Speicherbedarfs eines Objekts sizeof(Typ) oder sizeof Ausdruck</td></tr>
</table>

Reihenfolge der Operanden-Auswertung:

Die Reihenfolge ist undefiniert, bis auf vier Ausnahmen:

- Rechter Operand bei **&&** nur bewertet, wenn **linker Operand wahr** ist
- Rechter Operand bei **||** nur bewertet, wenn **linker Operand falsch** ist
- Beim Konditional-Operator **?:** wird zuerst der erste Operand ausgewertet
- Beim Komma-Operator "," wird zuerst der erste Operand ausgewertet

Folge:

- (a+b)+(c+d) legt nicht fest, ob der erste oder zweite geklammerte Summand zuerst ausgewertet wird.
- Unbeabsichtigte Nebeneffekte, z.B. ist bei

  ```
  f(&x) + g(&x)
  ```

  nichts darüber ausgesagt, ob zuerst **f()** oder zuerst **g()** die Variable x möglicherweise verändern (hängt von der Aufrufreihenfolge ab)!

Ausdrücke basieren auf Primärausdrücken.

Primär-Ausdrücke sind

- Konstanten (Wert der Konstanten)
- Konstante Zeichenfolgen (Zeiger auf das erste Element)
- Namen von Variablen, Funktionen oder Feldern (Wert der Variablen, Zeiger auf Funktion, Zeiger auf das erste Feldelement)
- Ausdrücke in Klammern (Wert und Typ des Ausdrucks)

Ausdrücke mit Verweisen auf Feld- oder Struktur-Elemente

Bildung aufgrund der Operatoren **[]**, ".”-Operator und "**->**":

- x[*ausdruck*]
- s.name
- ps->name, entspricht (*ps).name

Funktions-Ausdrücke (bedeutet Funktionsaufruf)

Bildung durch Angabe des Funktionsnamens und der Argumentliste:

x (argument_list)

Adress-Ausdrücke

- Operator **&**, anzuwenden auf ein adressierbares Objekt, liefert dessen Adresse
- Der Operator *****, anzuwenden auf einen Ausdruck vom Typ “Zeiger auf Objekt”, liefert den Wert das Objekt selber.

  Insbesondere gilt:

  ***(&x) identisch mit x**

sizeof-Ausdrücke

sizeof *ausdruck*:

der erforderlichen Bytes, um den Ausdruck abzuspeichern. Ist *ausdruck* ein Feldname, ergibt der **sizeof**-Ausdruck die Gesamtzahl der Bytes im Feld!

sizeof (typname):

Anzahl Bytes, die ein Objekt des Typs *typname belegt*.

Casts (explizite Typumwandlungen)

(*typname*)x wandelt den Operanden **x dynamisch in** den Typ ***typname***.

Beispiel:

```
int a=1,b=4;
double x = (double)a/b;          /* ergibt 0.25 */
x = a/b;                         /* ergibt 0 */
```

Arithmetische Ausdrücke

Gebildet durch

- Unäre Operatoren + und –
- Unäre Operatoren ++ und —
- Binäre Operatoren +,–,*,/,%
- Arithmetische Operanden

Ausdruck	Wirkung/Wert
+x	Unäres Plus (Vorzeichenoperator)
-x	Unäres Minus (liefert negativen Operandenwert)
x+y	Addition von x und y, mit Anpassung
x-y	Subtraktion von x und y, mit Anpassung
x*y	Multiplikation von x und y, mit Anpassung
x/y	Division von x durch y, mit Anpassung, ganzzahlig wenn beide Operanden ganzzahlig
++x	x wird um 1 erhöht. Ist x Zeiger, wird x um die Objektgröße inkrementiert. Danach wird x weiterverwendet.
--x	x wird um 1 erniedrigt. Ist x Zeiger, wird x um die Objektgröße dekrementiert. Danach wird x weiterverwendet.
x++	x wird ausgewertet. Danach x wird um 1 erhöht. Ist x Zeiger, wird x um die Objektgröße inkrementiert.
x--	x wird ausgewertet. Danach wird um 1 erniedrigt. Ist x Zeiger, wird x um die Objektgröße dekrementiert.

Beispiele:

```
int i=1,j;
j = ++i + 1;        /* i := 2, j := 3 */
i = 1;
j = i++ + 1;        /* i := 2, j := 2 */
```

Vergleiche und logische Ausdrücke

C hat folgende Vergleichsoperatoren					
>	**>=**	**<**	**<=**	**==**	**!=**

Der Vergleich ist zulässig zwischen **arithmetischen Operanden**(ggfls. mit Anpassung), sowie eingeschränkt auch zwischen **Zeigern**. Vergleiche bei **Zeigern** sind nur definiert bei Zeigern auf Teilobjekte **eines** Feldes oder **einer** Struktur (sonst undefiniert); verglichen werden hier die Abstände von der Basisadresse.

Vergleiche liefern als Ergebnis einen **logischen Ausdruck** (Typ **int**) mit dem Wert (Vergleich wahr) bzw. 0 (Vergleich falsch).

Logische Werte werden (im Normalfall) mit den Operatoren &&, || und miteinander verknüpft.

Ausdruck	Wirkung/Wert
a && b	1, wenn beide Operanden wahr sind, 0 sonst. b wird nur ausgewertet, wenn a wahr ist (Logisches UND).
a || b	1, wenn einer der beiden Operanden wahr sind, 0 sonst. b wird nur ausgewertet, wenn a falsch ist (Logisches ODER).
!a	1, wenn a falsch (== 0) ist, sonst 0 (Logische NEGATION).

Beispiel:

```
int main(int argc, char **argv)
{
    int res = 0;
    5<3 && res++ > 3;        /* res wird nicht erhoeht */
    ...
    5>3 && res++ > 3;        /* res wird auf 1 gesetzt */
    ...
}
```

Bit-Ausdrücke

Einzelne Bits können mit den Bit-Operatoren manipuliert werden:

Ausdruck	Wirkung/Wert
a << b	Links-Shift von a um b Stellen, kann als Multiplikation von a mit 2^b interpretiert werden
a >> b	Rechts-Shift von a um b Stellen, Vorsicht bei vorzeichenbehaf-tetem a! Division von a durch 2^b
a & b	Bitweises UND von a und b
a | b	Bitweises ODER von a und b
a ^ b	Bitweises EXKLUSIVES ODER von a und b
~b	Einerkomplement b (bitweise invertieren)

Beispiel:

```
unsigned char i = 1;
i <<= 3;            /* 8 */
i >>= 2;            /* 2 */
i |= 5;             /* 7 */
i &= 3;             /* 3 */
i ^= 5;             /* 6 */
i = ~i;             /* 249 */
```

Konditional- und Komma-Ausdrücke

Konditional-Ausdrücke werden durch den Operator ?: gebildet, Komma-Ausdrücke durch den Komma-Operator “,”.

Ausdruck	Wirkung/Wert
e0 ? e1 : e2	Ist der Ausdruck e0 wahr, hat der Gesamtausdruck den Wert von e1, ansonsten den Wert von e2. Im ersten Fall wird **e2 nicht bewertet**!
e1,e2	Zuerst wird e1 ausgewertet, dann e2. Der Gesamtausdruck hat e2 als Wert und den Typ von e2 als Typ. Vorsicht bei Funktionsaufrufen, wenn Komma-Ausdrücke auf Parameter-Position eingesetzt werden! Bitte klammern!

Beispiel:

```
int i,j,max = i>j ? i : j;        /* cond-Ausdruck */
f(100,(j=10,max=11));             /* Komma-Ausdruck, nicht */
f(100,j=10,max=11);
```

Konstante Ausdrücke

Konstante Ausdrücke sind zur Übersetzungszeit bekannt und müssen in folgendem Zusammenhang eingesetzt werden:

- Größen und Dimensionen von Feldern
- Auswertung von -Ausdrücken in -Anweisungen
- Länge von Bitfeldern in Strukturen/Unions
- Aufzählungswerte
- Initialisierung von static- und extern-Variablen

Operanden sind Integer- und Character-Konstanten, als Operatoren sind zulässig:

+ - ~ ! * / % << >> == != < <= > >= ? ^ | & || ?:

Zuweisungs-Ausdrücke

a *op* b
op∈{**=,*=,/=,%=,+=,-=,<<=,>>=,&=,^=,|=**}

Der linke Operand einer Zuweisung muß ein L-Value sein (also kein Vektor, keine Konstante, nicht const). Der Wert des Zuweisungs-Ausdrucks ist der Wert der rechten Seite.

Schreibweise ***op*=** ist abkürzend:

a *op*= b

die gleiche Bedeutung wie

a = a *op* b.

Nebeneffekte

Bei Berechnung eines Ausdrucks oder bei einem Funktionsaufruf wird der **Wert einer Variablen verändert**. C arbeitet also immer mit Nebeneffekten: Die Wertzuweisung ist also keine Anweisung, sondern ein Operator: es lassen sich die Werte von Variablen nur durch Nebeneffekte verändern!

Weitere Nebeneffekte durch Inkrement- und Dekrementoperatoren **++** und **--** sowie alle Funktionen mit "Ausgabeparametern" (wie z.B. **scanf**).

C-Standard zu Nebeneffekten:

- Nebeneffekte können nicht eintreten, bevor die Auswertung des Ausdrucks beginnt, durch die sie erzeugt werden.
- Wenn die Auswertung eines Ausdrucks abgeschlossen ist, müssen auch seine Nebeneffekte eingetreten sein.
- Wenn eine Variable innerhalb eines Ausdrucks mehrfach angesprochen wird und außerdem Nebeneffekte für sie eintreten, ist nicht definiert, welchen Wert die Variable im Einzelfall besitzt:

```
x[i] = ++i;
v[i] = w[i++];
s += v1[i] * v2[i++];
a = ++a % 8;
f(--i,++i);
/* welchen wert hat i auf den Parameterpositionen? */
```

7. Typumwandlungen

Implizite Typ-Umwandlungen sind nötig, wenn an einem Ausdruck **mehr als ein Typ beteiligt ist**. Zur Bewertung findet eine Umwandlung **aller Typen auf einen gemeinsamen** (durch den Compiler) statt. Daneben sind explizite Umwandlungen mittels des casts (*typname*) möglich. In den Tabellenspalten sind erlaubte Umwandlungen durch ein Pluszeichen markiert, verbotene sind nicht markiert.

Nach → Umwandlung Von ↓	void	integer	real	Pointer	Feld	Struktur	Funktion
void	+						
integer	+	+	+	+			
real	+	+	+			+	
Pointer	+	+		+	+		+
Feld	+					+	
Struktur	+						
Funktion	+						

Bei Typ-Umwandlungen werden ggfls. auch **Werte gewandelt** mit eventuellem **Informationsverlust** (z.B. **float** nach **int** - hier wird nur der Ganzzahl-Anteil übernommen)**.**
Arithmetische Umwandlungen sind i.A. unkritisch, wenn der *kürzere* Typ in den *längeren* gewandelt wird); siehe folgende gewichtete Aufstellung:

```
long double        /* Länger */
double
float
long unsigned
unsigned int
int                /* Kürzer */
```

Konversion innerhalb von Integer-Werten (z.B. bei arithemischen Verknüpfungen):

char, unsigned char, short	nach **int**
unsigned short	nach **unsigned int**, falls **short** eine Länge wie **int** besitzt, ansonsten nach **int**

Vorsicht: Die Implementierung des Zeichenwerts **char** ist **systemabhängig**!

Umwandlungen nach **void** z.B. sinnvoll, wenn der Rückgabewert einer Funktion ignoriert werden soll:

```
(void) printf(”%d”,x);
```

Umwandlung in Integer-Typen:

- Bei der Umwandlung von **Integer-Typen** in Integer-Typen wird entweder direkt umgewandelt oder abgeschnitten, oder das Ergebnis ist undefiniert.

- Wird ein **Gleitkomma-Typ** in einen Integer-Typ gewandelt, so wird entweder der ganzzahlige Anteil umgewandelt, oder, falls nicht als Integerwert darstellbar, ist das Ergebnis undefiniert.

- Zeigertypen werden in vorzeichenlose Zahlen umgewandelt .

Umwandlung von anderen Typen in Gleitkomma-Typen:

- Bei der Umwandlung von **Gleitkomma-Typen** in Gleitkomma-Typen wird entweder direkt umgewandelt oder gerundet oder abgeschnitten, oder das Ergebnis ist undefiniert.

- **Integer-Typ** in Gleitkomma-Typ: entsprechende Näherung.

Umwandlung in Zeiger-Typen:

- Umwandlung von **Integer-Typen** in **Zeiger–Typen** ist möglich, aber nicht portabel
- Umwandlung eines **Feld-Typs in einen Zeiger–Typ:** auf das erste Feld-element
- Umwandlung eines **Funktions-Typs in einen Zeiger-Typ**: Zeiger auf Funktion

Implizite Umwandlung bei Zuweisungen

In folgenden Fällen findet eine implizite Umwandlung der rechten Seite in den Typ der linken Seite zur Anpassung statt:

Linke Seite	Rechte Seite
arithmetischer Typ	arithmetischer Typ
Zeigertyp	Integer-Konstante 0
Zeiger auf Typ T	Feld von Typ T
Zeiger auf Funktion	Funktionstyp

Beispiele:

```
int i = 1L;          /* Umwandlung long int ->int */
int *iptr = 0;       /* Wert entspricht (int *)0 */
int ia[2];
iptr = ia;           /* Umwandlung nach int-Zeigertyp */
```

Implizite Umwandlung bei Funktionsaufrufen

In folgenden Fällen implizite Umwandlung zur Anpassung:

Ursprünglicher Typ	gewandelter Typ auf Parameterposition
char und short	int
unsigned char und unsigned short	unsigned int
float	double
Feld von Typ T	Zeiger auf T
Funktion von Typ T	Zeiger auf Funktion von Typ T

Typumwandlungen beim Auftreten binärer Operatoren

arithmetische Operanden verschiedene Typen, dann erfolgt vor der Ausführung der Operation eine Umwandlung in den *“größeren”* Typ der beteiligten Operanden. Bei Mischungen (integer mit Gleitkomma) werden zuvor die früheren Regeln angewandt.

- **long double**
- **Double**
- **float**

Sind alle Operanden Integer-Operanden, erfolgt die Konversion gemäß:

- Hat ein Operand den Typ **unsigned long int**, wird der andere ebenfalls nach **unsigned long int** umgewandelt.
- Falls **long int** mit **unsigned int** verknüpft wird, werden beide Operanden nach **long int** umgewandelt, falls dieser Typ alle unsigned int Werte darstellen kann – ansonsten nach **unsigned long int**.
- Ist ein Operand vom Typ **long int,** wird der andere ebenfalls nach **long int** umgewandelt.
- Ist ein Operand vom Typ **unsigned int,** wird der andere ebenfalls nach **unsigned int** umgewandelt.

Beispiel:

```
int i;
long int il
double d;
unsigned long int iul;

il = i + 1L;         /* Ausdruck hat Typ long int */
d = 1.0 + 1;         /* Ausdruck hat Typ double */
iul = iul+1;         /* Ausdruck hat Typ unsigned long int */
```

8. Anweisungen

Jedes C-Programm, genauer: jede **C-Funktion, besteht aus Anweisungen** (statements), in denen die dynamischen Aktionen festgelegt werden.

Eine Anweisung wird **durch “;” abgeschlossen** (im Gegensatz beispielsweise zu Perl, wo Anweisungen durch “;” voneinander getrennt werden).

Anweisungen werden **sequentiell abgearbeitet**, solange dieser Fluss nicht anderweitig durch Sprünge, Schleifen oder bedingte Anweisungen unterbrochen wird.

Einfache Anweisungen

Einfache Anweisungen unterbrechen nicht den sequentiellen Programmfluss.
Sie werden unterteilt in:

Leere Anweisung: ;

Wird überall dort eingesetzt, wo eine Anweisung stehen muss, aber keine Aktion erforderlich ist.

Beispiel:

```
int i;
for (i=0; i<10; i++) ;       /* Leere Anweisung in for-Schleife */
```

Ausdrucks-Anweisung: *ausdruck***;**

Anwendung: Zuweisungen oder Funktionsaufrufe.

Beispiel:

```
int i;
i = 1;        /* der Ausdruck "i = 1" als Anweisung */
5;            /* Die Konstante 5 als Anweisung */
```

Labels: *label: anweisung***;**

ist ein Bezeichner, gefolgt von einem Doppelpunkt. Ein Label kann als Sprungziel in einer **goto**-Anweisung verwendet werden (**die in den allermeisten Fällen überflüssig ist**!!). Der Gültigkeitsbereich erstreckt sich auf den umgebenden Block.

Beispiel:

```
    fehler:
            return 0;
```

Anweisungsblöcke:

```
{
    deklaration;...
    anweisung; ...
}
```

Ein Anweisungsblock besteht aus einer oder mehreren Definitionen und Deklarationen, aus einer oder mehreren Anweisungen, die insgesamt von einem Klammernpaar {} umgeben sind.

Syntaktisch zählt ein Block als **eine Anweisung**. Folge: ein Block darf überall dort stehen, wo eine Anweisung stehen darf. Im Deklarationsteil angegebene **automatische Variable** werden bei dem Eintritt in den Block **stets neu angelegt**. **Jede Variable**, die in einem übergeordneten Block angelegt wird, ist auch in **nachgeordneten Blöcken gültig**, soweit nicht redefiniert.

Eine **Funktion besteht aus mindesten einem Block**; den obersten nennt man auch **Funktionsrumpf.**

Beispiel:

```
#include <stdio.h>
int main(int argc, char *argv[])
{
    int i = 1,
        j = 1;
    {
        int i = 1;                  /* lokales i */
        i++;                        /* lokales i inkrementiert
        j++;                        /* übergeordnetes j inkrementiert */
    }
    printf(%d %d\n",i,j);           /* i hat Wert 1, j den Wert 2 */
}
```

Nur zur Demonstration, da schlechter Programmierstil!

Schleifen - while-Schleife: while (*ausdruck*) *anweisung*

Solange der Ausdruck *ausdruck* einen von 0 verschiedenen Wert aufweist, wird die Anweisung *anweisung* ausgeführt. *ausdruck* wird vor jedem Schleifendurchlauf mit allen Seiteneffekten neu bewertet, also auch am Schleifenbeginn.

Hat also *ausdruck* zu Beginn den Wert 0, wird die Schleife nicht durchlaufen (Eine while-Schleife ist eine **abweisende** Schleife).

Beispiel:

```
main()
{
        char *s = "0123456789";
        int i = 0;
        while (s[i])            /* Abbruch bei \0, String-Ende */
        {
                i++;
        }
        printf ("%d\n",i);      /* gibt laenge von s aus */
}
```

Schleifen - do-while-Schleife: do *anweisung* while (*ausdruck*)

Solange der Ausdruck *ausdruck* **nach einem Schleifendurchlauf** einen von 0 ver schiedenen Wert aufweist, wird die Anweisung *anweisung* ausgeführt. *ausdruck* wird nach jedem Schleifendurchlauf mit allen Seiteneffekten neu bewertet. Die Schleife wird **mindestens einmal durchlaufen**.

Beispiel:

```
main()
{
        char *s = “0123456789”;
        int i = -1;
        do                              /* Abbruch bei \0, String-Ende */
        {
                i++;
        }
        while (s[i] != ’\0’);
        printf (”%d\n”,i);          /* gibt laenge von s aus */
}
```

Schleifen - for-Schleife:

for (*$ausdruck_1$*;*$ausdruck_2$*;*ausdruck3*) *anweisung*

Die Schleife ist vollkommen äquivalent zu:

```
ausdruck1;
while (ausdruck2)
{
    anweisung
    ausdruck3;
}
```

(solange keine **continue**-Anweisung im Schleifenrumpf enthalten ist). ***$ausdruck_1$*** ist der **Schleifen-Initalisierungsausdruck**, ***$ausdruck_2$*** wird als Bedingungsausdruck vor je-dem Schleifendurchlauf bewertet (mit Abbruch, wenn logisch falsch), und ***$ausdruck_3$*** nach jedem Schleifendurchlauf.

Beispiel:

```
main() /* Addiert die Zahlen von 1 bis 10 */
{
    int i,
            sum,
            n = 10;
    for (i=1,sum=0; i<=n; i++)
    {
        sum += i;
    }
}
```

Sprunganweisungen: break-Anweisung

Anweisung bewirkt das sofortige Verlassen der innersten umgebenden Schleife einer **for**-, **do**-, **while**- oder **switch**-Anweisung.

```
#define TRUE 1
main()
{
        char *s = "0123456789";
        int i = 0;
        while (TRUE)
        {
              if (!s[++i]) break;          /* abbruch, wenn String-Ende */
        }
        printf (”%d\n”,i);                 /* gibt laenge von s aus */
                                           /* Wert von i: Stringlänge */
}
```

Sprunganweisungen: continue-Anweisung

Diese Anweisung bewirkt den sofortigen Einstieg in den nächsten Durchlauf der innersten umgebenden Schleife einer **for**-, **do**-, **while**-Anweisung.

```
main()      /* zählt alle Zeichen ungl. 'i' in einem String */
{
        char *s = "Zeichenfolge mit i-Zeichen";
        int i = 0, laenge = 0;
        while (s[i])
        {
                if (s[i++] == 'i') continue;
                ++laenge;
                /* oder: if (s[i++] != 'i') ++laenge; statt der beiden oberen Zeilen */
        }
}
```

Sprunganweisungen: return *ausdruck*$_{opt}$;

Anweisung bewirkt das Verlassen einer Prozedur. Bei Funktionen ist ein Ausdruck mit dem Rückgabewert anzugeben:

```
double max(double a,double b) /* berechnet das Maximum zweier Zahlen */
{
    return a>b ? a:b;
}
```

Sprunganweisungen: goto *label*;

Die Anweisung **goto ist im Prinzip nicht erforderlich**. Sie erlaubt das Springen zu einer anderen Stelle im Programm:

```
for (...)
{
        while (...)
        {
                ...
                if (ausnahmebedingung) goto fehler_abbruch;
        }
}
...
fehler_abbruch: return 99;
```

Bedingungs-Anweisung: if (*ausdruck*) *anweisung*;

Ausführung der Anweisung *anweisung* wird vom booleschen Wert des Ausdrucks *ausdruck* abhängig gemacht, d.h. die Ausführung erfolgt nur, wenn die Bewertung von *ausdruck* einen Wert $\neq 0$ ergibt:

```
if (a > 0)
    b = log(a);
```

Bedingungs-Anweisung: **if (*ausdruck*) *anweisung_1*;**
else *anweisung_2*;

Die Ausführung der Anweisung *anweisung_1* wird vom booleschen Wert des Ausdrucks *ausdruck* abhängig gemacht, d.h. die Ausführung erfolgt nur, wenn die Bewertung von *ausdruck* einen Wert $\neq 0$ ergibt. Andernfalls wird Anweisung ***anweisung_2*** ausgeführt.

```
if (a > 0)
    b = log(a);
else
    printf(”can’t compute logarithm of %g\n”,a);
```

Bedingungs-Anweisung: **if (*ausdruck_1*) *anweisung_1*;**
else *if (ausdruck_2*) *anweisung_2*
...;
else *anweisung_n*;

Die Ausführung der Anweisung *anweisung_i* wird vom booleschen Wert des Ausdrucks *ausdruck_i* abhängig gemacht, d.h. die Ausführung erfolgt nur, wenn die Bewertung von *ausdruck_i* einen Wert ≠0 ergibt. Andernfalls wird in der nächsten "**else if**"-Anweisung Ausdruck *ausdruck_i+1* überprüft usf.

Wenn *ausdruck_i* wahr ist, werden die restlichen Ausdrücke nicht mehr bewertet und die Anweisungen **nicht mehr ausgeführt** !!

```
if (a > 0)
     b = sqrt(a);
else if (a == 0)
     b = 0;
else
     b = sqrt(-a);
```

Switch-Anweisung: **switch (*ausdruck*)**

{ case konstante$_1$: anweisung$_1$;

case konstante$_2$: anweisung$_2$;

...

default: anweisung$_n$;

}

Der Ausdruck *ausdruck* in der **switch**-Anweisung wird bewertet - er muss einen Integer-Wert liefern. Danach werden die einzelnen **case**-Marken abgefragt: stimmt der Ausdruck ausdruck mit einer der ganzzahligen Konstanten konstante$_i$ überein, wird Anweisung Nummer. i ausgeführt und ggfls. die Anweisung hinter der nächsten **case**-Marke ausgeführt. Ist dies nicht gewünscht, muss anweisung$_i$ mit einer **break**-Anweisung enden.

Stimmt *ausdruck* mit keiner der Konstanten konstante$_i$ überein, wird die Anweisung bei der **default**-Marke, falls vorhanden, ausgeführt.

Die Konstanten konstante$_i$ müssen unterschiedlich sein.

Beispiel:

```
main()     /* zählt, wie oft Ziffern in einem String vorkommen */
{
    char *s = "Er gewann in der 37. Woche 123.456,00 EURO im Lotto";
    int i, lenx = 0;
    for (i=0; s[i]; i++)
    {
        switch(s[i])
        {
            case '0' :
            ...
            case '9' : ++lenx; break;
            default:
        }
    }
    printf("Der String enthält %d Ziffern\n",lenx); /* 10 */
    return 0;
}
```

9. Funktionen

Jedes C-Programm besteht aus **mehreren C-Funktionen**, die - aus Gründen der Modularität - für die verschiedensten Aufgaben zuständig sind.

Achtung: zwischen K&R-C und ANSI-C bestehen teilweise gravierende Unterschiede zwischen den Formaten von Definition und Deklarationen.

Die main-Funktion

Eine C-Funktion ist in jedem C-Programm **zwingend erforderlich: die main-Funktion**, die beim Start des Programmes durch das Betriebssystem als erstes aufgerufen wird.

```
int main(int argc, char *argv[])
{
        anweisungen des hauptprogrammes ...
        return ausdruck;
        /* exit-status; das, was die UNIX-Shell als $? liefert */
}
```

argc enthält die Anzahl der übergebenen Programm-Optionen inkl. Programm-Name, und **argv** ist ein Feld von Strings, das die einzelnen Programm-Optionen enthält.

```
int main(int argc, char *argv[])
{
    int i;
    for (i=0; i<argc; i++) printf("%d-ter Parameter \
    %s\n",i,argv[i]);
    return 0;
}
```

Funktionen als Programmeinheiten

Durch Funktionen sind Programme **in kleinere Einheiten zerlegbar**; im Allgemeinen in viele kleine und wenige große (soweit möglich).

Dem Compiler müssen alle verwendeten Funktionen bekannt sein. Eine Funktion muss also zumindest deklariert sein, bevor sie aufgerufen werden kann.

Die **Definition kann** in einer **gesonderten Quelldatei**-externe Funktion oder in der gleichen Datei hinter dem Funktionsaufruf erfolgen.

Im folgenden Beispiel wird in **main()** eine Funktion **flmax** verwendet:

<table>
<tr><th>Variante 1</th><th>Variante 2</th></tr>
<tr><td>
<pre>
float flmax(float a,float b)
{
        return a>b?a:b;
}
init main()
{
        float x,y,z;
        x=5;
        y=3.14;
        z=flmax(x,y);
        return 0;
}
</pre>
</td><td>
<pre>
int main()
{
        float flmax(float,float),
              x=5,
              y=3.14,
              z;
        z=flmax(x,y);
        return 0;
}
float flmax(float a,float b)
{
        return a>b?a:b;
}
</pre>
</td></tr>
</table>

Bei umfangreicheren Programmen hält man üblicherweise die **Funktions-Definitionen** in C-Sourcen (**funktion.c**), und die **Funktions-Deklarationen** in C-Headern (**funktion.h**); nur letztere werden vom Anwender includiert:

Variante 3		
main.c	**flmax.h**	**flmax.c**

include "flmax.h"
init main()
{
float x,y,z;
x=5;
y=3.14;
z=flmax(x,y);
return 0;
}
	float flmax(float,float);	float flmax(float a,float b)
		{
		return a>b?a:b;
		}

Neben den vom Programmierer definierten Funktionen gibt es **vorübersetzte Standardfunktionen**, die ebenfalls mithilfe von Header-Dateien deklariert werden, wie z.B.

```
#include <stdio.h> (oder #include “stdio.h”)
```

Diese Standardfunktionen liegen in vorübersetzer Form in **Bibliotheks-dateien** vorund werden vom **Linker mitangebunden** (unter UNIX: **libC.a**, **libC.so** u.ä.).

Rückgabewerte von Funktionen

Außer bei Prozeduren, die definitionsgemäß keine Werte zurückgeben - in C ist dann der Return-Typ **void** - gibt jede Funktion einen Wert zurück, und zwar als **Argument** der **return**-Anweisung.

Fehlt eine return-Anweisung und wird in der Funktion die letzte Anweisung erreicht, ist der **Rückgabewert undefiniert**. Dies ist eine häufige Fehlerquelle. **Gute Compiler** erkennen diesen Sachverhalt und **geben eine Warnung aus**.

Ist man am Rückgabewert einer **Funktion** nicht interessiert, sollte man sie **als Prozedur** aufrufen: dazu dient der Cast (**void**) vor dem Funktionsaufruf.

Parameterübergabe an Funktionen

In C werden die **Parameter** bei der Übergabe an eine Funktion in einen Zwischenspeicher (**Stack**) **kopiert** und von dort aus von der Funktion verarbeitet. **Änderungen an Übergabe-Parametern** wirken sich also nur **innerhalb der Funktion** (im Stack) aus, aber nicht in der aufrufenden Funktion.

```
float mul(float a, float b)
{
        a = a*b; return a;
}

int main()
{
        float x,y,z;
        x = 5;
        y = 3.14;
        z = mul(x,y);      /* x bleibt unverändert */
}
```

Umgehung durch Übergabe von Zeigern (oder durch Verwendung **externer** oder dateiglobaler **Variablen)** statt Übergabe der Originale an die Funktion. **Standardmäßig** geschieht dies bereits bei **Arrays und Strings**, bei denen aus Effizienzgründen lediglich ein Zeiger übergeben wird und damit automatisch die Array-Elemente modifizierbar sind.

Bei Nichtbeachtung der obigen Regeln ergeben sich leicht Semantikfehler. Beispiel: das Vertauschen zweier Variable durch einen Funktionsaufruf:

<table>
<tr><th>Falsche Variante</th><th>Korrekte Variante</th></tr>
<tr><td>

```
void swap(float a,float b)
{
        float x=a;
        a=b;
        b=x
}
init main()
{
        float x,y;
        x=5;
        y=3.14;
        swap(x,y)
}
```

</td><td>

```
void swap(float *a,float *b)
{
        float x=*a;
        *a=*b;
        *b=x
}
init main()
{
        float x,y;
        x=5;
        y=3.14;
        swap(&x,&y);
}
```

</td></tr>
</table>

Vorsicht bei der **Übergabe von Feldern;** da nur ein Zeiger auf das erste Feldelement übergeben wird, ergeben sich leicht **fehlerhafte Zugriffe** durch **Feldgrenzen-Überschreitung**, die der Compiler nicht überprüfen kann:

```
void quad(int *,int);
int main()
{
        int a[10] = {1,2,3,4,5,6,7,8,9,10};
        quad(a,10);               /* allgemeiner : sizeof a / sizeof (int) */
}

void quad(int *feld,int len)      /* len Elemente quadrieren */
{
        for (;len>0; len--,feld++) (*feld) *= (*feld);
                                  /* beachte: feld wächst bei Inkrementierung */
                                  /* um Länge von int */
}
```

Zeichenfolgen stellen einen **Spezialfall von Feldern** dar, werden also genauso behandelt. Der Abschluss von Strings mit einem NULL-Character **vereinfacht** jedoch die **Parameterübergabe** (Längenangaben können teilweise entfallen).

Ist das Feld **a** mehrdimensional, kann im Funktionskopf von **quad** die Dimensionsangabe bei der ersten Dimension entfallen:

```
#include <stdio.h>
void quad(int [][5],int,int);
int main()
{
        int a[10][5];
        int i,j;
        for (i=0; i<10; i++) for (j=0; j<5; j++) a[i][j] = 10*i+j;
        quad(a,10,5);
}
void quad(int m_feld[][5],int len1, int len2)
/* m_feld quadrieren */
{
        for (;len1>0; len1--)
        {
              int len22 = len2;
              for (;len22>0; len22--)
              {
                      m_feld[len1-1][len22-1] *= m_feld[len1-1][len22-1];
              }
        }
}
```

Statische und externe Variable in Funktionen

Jede lokale Funktions–Variable, für die nichts anderes angegeben ist, wird **bei jedem Funktionsaufruf neu initialisiert**. Dies kann man verhindern, indem eine Variable als **static** klassifiziert wird.

```
main()
{
        int i;
        for (i=0;i<10;i++) counter();
}

void counter(void)    /* gibt aus, wie oft aufgerufen */
{
        static int c = 0;
        printf("$d. Funktionsaufruf\n",++c);
}
```

10. Ein-/Ausgabe

Problem: **Wie tritt ein C-Programm mit der Außenwelt in Kontakt?**

Außenwelt:

- **Bildschirm**
- **Tastatur**
- **Dateien**
- ...

Antwort: Die Programmiersprache C bietet dafür **keine Unterstützung**!, im Gegensatz z.B. zu Fortran90/95, PL/I, Pascal,...

Aber: Es gibt Bibliotheken mit Funktionen zur Kontakt-Aufnahme mit der Außenwelt. Eine davon, eine der wichtigsten, ist die Standard-E/A-Bibliothek. Sie wird mit

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

in das Programm eingebunden.

Ein-/Ausgabe über die Standard-E/A-Einheiten (Standardeinheiten)

Standardeinheiten sind (beim Arbeiten mit einem Terminal)

stdin: entspricht der Tastatur
stdout: entspricht dem Bildschirm
stderr: entspricht dem Bildschirm

Umlenkungen auf andere Medien sind möglich, unter UNIX z.B. mit einer Kommando-Shell.

Funktionen zum Ansprechen der Standardeinheiten	
putchar(c)	Ausgabe eines Zeichens auf stdout
printf("format",arg1,arg2,...)	Formatierte Ausgabe mehrerer Ausdrücke auf stdout
puts(str)	Ausgabe eines Strings (Zeichenkette) aus stdout
getchar()	Einlesen eines Zeichens von stdin
gets()	Einlesen einer Zeichenfolge von stdin
scanf("format",parg1,parg2,...)	Einlesen verschiedener Variabler verschiedenen Typs von stdin

Zeichen ein- und ausgeben:

```
int char getchar(void);
int putchar(int c);
```

liest von **stdin** bzw. schreibt ein Zeichen nach **stdout** und liefert es als Funktionswert zurück.

Bei Fehlern oder Dateiende (z.B. mit Ctrl D, oder kein Platz mehr) wird dagegen EOF zurückgemeldet. EOF wird in **stdio.h** definiert. Beispiel:

```
#include <stdio.h>
...
while (1)
{
        int c;
        if ((c = getchar()) == EOF) break;
        /* Einlesen bis Eingabeende */
        putchar(c);                         /* Zeichen ausgeben */
        ...
}
```

Komplette Zeilen lassen sich mit **gets** von einlesen:

```
char *gets(char *string);
```

String:
Zeiger auf Character-Bereich (**char**-Feld sein, oder dynamisch allokiert), in den eingelesen wird, bis Zeilenende erkannt wird.

Zeilenende-Zeichen:
wird nach Einlesen durch das Zeichen **’\0’** ersetzt.

Fehlerfall oder Dateiende:
NULL (0-Pointer) als Rückgabe

Normalfall:
string als Rückgabewert. **gets** ist dadurch in andere Funktionsaufrufe einsetzbar.

Achtung: gets hat **Sicherheits-Schwächen**:

Wenn eine Eingabezeile länger als die Länge von s ist, werden möglicherweise Programmteile undefiniert überschrieben mit der Folge eines Crash's oder eines anderen ungewollten Verhaltens. Ursache teilweise auch darin begründet, daß es bei C **keine Bereichsüberprüfung zur Programmlaufzeit** gibt!

Besser: **fgets** verwenden!

Eine **komplette Zeile** lässt sich mit der Routine puts auf stdout ausgeben:

```
int puts(char *s);
```

Dabei wird das '\0'-Character am Ende des Character-Strings bei der Ausgabe durch ein Newline-Zeichen ersetzt.

Beispiel für die Verwendung von `gets/puts`:

```
#include <stdio.h>
int main(int argc,char **argv)
{
	char *s = (char *) malloc(80),
	          *t = s;
	int l;
	if (s == NULL) return 1;	/* Fehler bei Speicherallokierung */
	while (t != NULL)
	{
		t = gets(s);		/* hoffentlich s nicht zu kurz! */
		if (t != NULL)		/* kein Dateiende */
		{
			l = 0;
			while (*t) l++,t++;	/* Laenge ausrechnen */
			...
			puts(s);		/* Eingabe spiegeln */
		}
	}
}
```

Ein-/Ausgabe mit Formatierung

Die Leistungen von **getchar/putchar/gets/puts** sind im allgemeinen nicht ausreichend, insbesondere beim Arbeiten mit numerischen Variablen. Zur formatierten Ausgabe solcher Variablen (und auch mehr) dient die Funktion .

```
int printf(char *format,arg1,arg2,...)
```

Rückgabewert: Anzahl ausgegebene Zeichen.

Im String **format** fortlaufend Angaben, in welchem Format die Variablen arg_i auszugeben sind.

Formatangabe

Eingeleitet mit dem Zeichen %, beendet mit einem Buchstaben, der den Typ der Variablen widerspiegelt. Außerhalb dieser Klammerung stehende Zeichen werden unverändert ausgegeben.

Buchstabe	Argumenttyp	Wandlung nach
d,i	int	Integer
o	int	Oktal, vorzeichenlos
x,X	int	Hexadezimal, vorzeichenlos
u	int	Integer, vorzeichenlos
c	int	Zeichen
s	char *	String, bis '\0', oder maximale Feldweite
f	double	Reelle Zahl im Format [-]mmm.ddd, standardmäßig 6 Nachkommastellen
e,E	double	Reelle Zahl im Format [-]m.dddexx, standardmäßig 6 Nachkommastellen
g,G	double	Benutze %e-Format, wenn Exponent <-4, bzw. vorgegebene Genauigkeit, ansonsten %f
p	void *	Zeigerwert, implementierungsabhängig
%		Keine Umwandlung, % selber ausgeben

Zwischen %-Zeichen und Formatbuchstabe sind weitere Spezifikationen (auch gemischt) möglich:

String	Wirkung
-	(Minuszeichen). Ausgabe des Resultates linksbündig (Standard ist rechtsbündig)
int-Zahl	Minimale Feldweite der Ausgabe
.int-Zahl	Maximale Feldweite der Ausgabe (bei reellen Zahlen Nachkomma-stellen)
h oder l	Darstellung von Integer-Zahlen als short- bzw. long-Wert

Der Ausgabe mit **printf** entspricht bei der Eingabe die Funktion :

$$\text{int scanf(char *format,parg}_1\text{,parg}_2\text{,...)}$$

scanf liest ab gemäß Formatangaben von der Standardeingabe, interpretiert fortlaufend die Formatangabe mit Konvertierung der eingelesenen Teilstrings in den angegebenen Typ und speichert das Ergebnis in den einzelnen Variablen über deren Pointer **parg¡** ab.

Beendigung des Einlesens bei **abgearbeitetem Formatstring** oder **un-passenden Eingabedaten**. Returnwert von **scanf** ist die Anzahl eingelesener Zeichen.

Besonderheiten:

- Leerzeichen und Tabs und Zeilenwechsel (**Whitespace**) werden überlesen
- Zeichen außerhalb der Formatangaben müssen auch in der Eingabe auftauchen, Sonderregel für Whitespace: Sie stehen für 0 bis mehrere Whitespaces und dienen als Trennzeichen.
- Formatangabe beginnt mit % und wird mit dem Formatbuchstaben abgeschlossen. Dazwischen optional:
  - Ganze Zahl als maximale Feldweite
  - h, l oder L (Konvertierung auf short/long)
  - Ein *-Zeichen. Bewirkt, dass die Eingabe interpretiert, aber nicht einer Variablen zugewiesen wird (”**überlesen**”).

Buchstabe	Argument-Typ	Ausgangsformat
d	int *	Integerzahl
i	int *	Integerzahl (auch oktal oder hexadezimal)
o	int *	Oktalzahl, vorzeichenlos
x,X	int*	Oktalzahl, vorzeichenlos
u	unsigned int *	Integerzahl, vorzeichenlos
c	char *	Zeichen (auch Leerzeichen oder TAB!)
s	char *	String ohne Leerzeichen oder TAB's. Die Variable muss ausreichend Platz bieten einschließließlich des angefügten '\0'
efg	float *	Reelle Zahl mit optionalem Vorzeichen, optionalem Dezimalpunkt und optionalem Exponenten
%		Das %-Zeichen selber

Beispiel: Eingabedaten für das folgende Programm:

"Zeichenfolge   1234 12.121314"

```
#include <stdio.h>
int main(int argc,char **argv)
{
      char s[100];
      int i;
      float x;

      scanf("%s %d %f",s,&i,&x);
      printf("String: %s; Real: %f; ganz: %10d\n",s,x,i);
      return 0;
}
```

Ausgabe:

```
String: Zeichenfolge; Real: 12.121314; ganz:       1234
```

Ein-Ausgabe mit Dateien

Bisherige Ein-/Ausgabe-Dienste sind ein Spezialfall für die allgemeine Datei-Ein/Ausgabe.

stdio.h enthält weitere standardisierte Dienste zum Arbeiten mit Dateien, die **unabhängig von den Medien** und **portabel** sind!

Zugriff über sogenannte **Streams.** Die Stream-E/A verwendet intern die nicht portablen System-Aufrufe. Streams arbeiten mit automatisch angelegten und verwalteten **Dateipuffern**, die für Programmierer transparent sind.

Streams werden über Zeiger auf spezielle Strukturen, den sogenannten **FILE-Strukturen,** realisiert. Ihre Inhalte sind für den Programmierer irrelevant und transparent.

Spezielle Streams sind die Standard–Ein, –Aus und –Fehlerausgabe. Sie werden durch **stdio.h** vordefiniert und sind bei Programmstart automatisch verfügbar und geöffnet:

```
FILE *stdin, *stdout, *stderr;
```

Öffnen und Schließen von Dateien

Zugriffe auf Dateien erfordern vor der eigentlichen Bearbeitung ein Öffnen der Datei zur Verknüpfung mit einem **FILE**-Zeiger. Dieser Zeiger identifiziert während der Bearbeitung die Datei.

Das Schließen erfolgt ebenfalls über den **FILE** -Zeiger.

```
FILE *fopen(char *filename,char *open_mode);
int fclose(FILE *stream); /* Rückgabewert: int */
```

Datei-Eröffnungsmodus		
"r"	Read	Eine bereits existierende Datei zum Lesen öffnen
"w"	Write	Datei zum Schreiben öffnen; alte Inhalte gehen verloren; Datei wird gegebenfalls angelegt
"a"	Append	Datei zum Schreiben am Dateiende öffnen; Datei wird gegebenenfalls neu angelegt
"r+"	Update	Bestehende Datei zum Lesen und Schreiben öffnen
"w+"	Read + Write, Datei neu erstellen	Datei zum Lesen und Schreiben öffnen, ggflls. Inhalt löschen.
"a+"	Read und Write	Datei zum Lesen und Schreiben am Dateiende öffnen, bei Nichtexistenz anlegen

Beispiel:

Öffne eine Datei, deren Name als Kommandozeilenparameter angegeben wird.

```
#include <stdio.h>
int main(int argc, char *argv[])
{
	FILE *stream = fopen(argv[1],"r");
	if (stream == NULL)
	{
		printf("Fehler: Datei %s kann nicht geoeffnet werden\n",argv[1]);
		return 1;
	}
	fclose(stream);
	return 0;
}
```

Beachte:

Bei Programmende erfolgt automatisch ein **fclose** aller offenen Dateien. Insofern ist **fclose(stream)** nicht erforderlich, aber **guter Programmierstil**.

Lesen und Beschreiben von Dateien

Im Prinzip geschehen Lesen und Beschreiben von Dateien ähnlich wie bei der Manipulation der Standard-Ein/Ausgabe, die letztendlich nur einen Spezialfall darstellt:

Funktionsname	**Prototype**	**Bedeutung**
fputc	int fputc(int,FILE *)	Schreibt einzelnes Zeichen
fputs	int fputs(char *,FILE *)	Schreibt Zeichenfolge
fprintc	int fprintf(FILE *,char *f,arg_i,...)	Schreibt Variablen formatiert
fgetc	int fgetc(FILE *)	Liest einzelnes Zeichen
fgets	char *fgets(char *,int N,FILE *)	Liest Zeichenfolge
fscanf	int fscanf(FILE *,char *f,$parg_i$,...)	Liest Zeichenfolge

Insbesondere ist beispielsweise

```
printf(format,arg1,...)
```

gleichwertig mit

```
fprintf(stdout,format,arg1,...)
```

Es gibt weitere vergleichbare, ähnlich benannte Routinen, die statt *Dateien* zu Lesen und zu beschreiben, *Strings* lesen und beschreiben (**sscanf** und **sprintf**).