

# Einführung: Editieren und Übersetzen

Um ein lauffähiges C-Programm zu erzeugen, muss zunächst mit einem Texteditor eine Quelltext-Datei angelegt und mit Code gefüllt werden. Anschließend wird ein Compiler gestartet, der den Quellcode in Maschinen-Code übersetzt und ein lauffähiges Programm erstellt.

Als klassisches Beispiel soll hierzu ein minimales Programm dienen, das lediglich "Hallo, Welt!" auf dem Bildschirm ausgibt. Hierzu wird mit einem Texteditor folgender Code in eine (neue) Datei `hallo.c` geschrieben:

```
// Datei: hallo.c          /* 1. */  
  
#include <stdio.h>           /* 2. */  
  
void main()                  /* 3. */  
{  
    printf("Hallo, Welt!\n"); /* 4. */  
}
```

Das obige Programm enthält folgende Komponenten:

1. Eine mit `//` eingeleitete Zeile am Dateianfang stellt einen Kommentar dar. Sie wird beim Übersetzen durch den Compiler ignoriert und dient lediglich der besseren Lesbarkeit. Ebenso werden Textbereiche, die durch `/*` und `*/` begrenzt sind, als Kommentare für Erklärungen oder Hinweise genutzt.<sup>1</sup>
2. Mit der Anweisung `#include <stdio.h>` wird dem Compiler mitgeteilt, die Standard-Input-Output-Bibliothek `stdio.h` zu laden.<sup>2</sup> Diese von vielen C-Programmen genutzte „Sammlung“ an Quellcode stellt u.a. Funktionen für die Ausgabe von Text auf dem Bildschirm bereit.
3. Die Funktion `main()` startet das Hauptprogramm, das sich innerhalb der folgenden geschweiften Klammern befindet. Jedes C-Programm verfügt über eine derartige `main()`-Funktion.<sup>3</sup>

<sup>1</sup> In vielen Programmen werden ausschließlich Kommentare verwendet, die mit den Zeichenfolgen `/*` und `*/` begrenzt sind. Hierdurch wird eine Kompatibilität mit alten C-Compiler-Versionen sicher gestellt. Im obigen Tutorium wird hingegen – nach persönlichem Geschmack – die `//`-Variante für (einzelige) Kommentare verwendet.

Zusätzliche Kommentare der Form `/* 1. */` dienen in diesem Tutorium als Marker, um im Text auf die jeweiligen Stellen im Quellcode eingehen zu können.

<sup>2</sup> Genauer gesagt gilt die Anweisung dem Präprozessor, einem Teil des Compilers.

<sup>3</sup> Die Bezeichnung `void` besagt lediglich, dass die Funktion keinen Rückgabe-Wert liefert, der anderweitig im Programm zu verwenden wäre.

4. Durch den Aufruf der Funktion `printf()` wird auf dem Bildschirm der in doppelten Hochkommata stehende Text ausgegeben. Die Zeichenfolge `\n` steht dabei als Zeichen für eine neue Zeile. Der Aufruf der Funktion muss, wie jede C-Anweisung, mit einem Strichpunkt ; beendet werden.

Um die Datei in lauffähigen Maschinen-Code zu übersetzen, wechselt man in einer Shell in den Ordner der Quellcode-Datei und ruft den Compiler `gcc` auf:

```
gcc hallo.c -o hallo
```

Durch die Option `-o hallo` wird dabei die Output-Datei, d.h. das fertige Programm, mit `hallo` benannt. Ist der Compilier-Vorgang abgeschlossen, kann das neu geschriebene Programm im gleichen Ordner aufgerufen werden:

```
./hallo  
# Ergebnis: Hallo, Welt!
```

Damit ist das erste C-Programm fertig gestellt. In den folgenden Abschnitten werden weitere Eigenschaften und Funktionen der Programmiersprache C erläutert sowie einige nützliche Werkzeuge und Programmiertechniken vorgestellt.

# Definition von Variablen

Ein wesentlicher Vorteil eines Computer-Programms gegenüber einem Taschenrechner besteht darin, dass es (nahezu beliebig viele) Werte und Zeichen in entsprechenden Platzhaltern („Variablen“) speichern und verarbeiten kann.

Da ein Computer-Prozessor nur mit Maschinencode arbeiten kann, müssen intern sowohl Zahlen wie auch Text- und Sonderzeichen als Folgen von Nullen und Einsen dargestellt werden. Dies ist aus der Sichtweise eines Programmierers zunächst nur soweit von Bedeutung, als dass er wissen muss, dass ein und dieselbe Folge von Nullen und Einsen vom Computer wahlweise als Zeichen oder als Zahl interpretiert werden kann. Der Programmierer muss dem Computer somit mitteilen, wie der Inhalt einer Variable zu interpretieren ist.

## Deklaration, Definition, Initialisierung

Um Variablen benutzen zu können, muss der Datentyp der Variablen (z.B. `int` für ganze Zahlen) dem Compiler mitgeteilt werden („Deklaration“). Muss dabei auch Speicherplatz reserviert werden (was meist der Fall ist, wenn sich die Deklaration nicht auf Variablen externer Code-Bibliotheken bezieht), so spricht man von einer Definition einer Variablen.

In C werden Variablen stets zu Beginn einer Datei oder zu Beginn eines neuen, durch geschweifte Klammern begrenzten Code-Blocks definiert. Sie sind im Programm gültig, bis die Datei beziehungsweise der jeweilige Code-Block abgearbeitet ist.<sup>1</sup>

Eine Definition von Variablen erfolgt nach folgendem Schema:

```
int n;
```

Es dürfen auch mehrere gleichartige Variablen auf einmal definiert werden; hierzu werden die einzelnen Namen der Variablen durch Kommata getrennt und die Definition mit einem abschließenden Strichpunkt beendet.

```
int x,y,z;
```

Wird einer Variablen bei der Definition auch gleich ein anfänglicher Inhalt („Initialwert“) zugewiesen, so spricht man auch von einer Initiation einer Variablen.<sup>2</sup>

<sup>1</sup> Die einzige Ausnahme bewirkt hierbei das Schlüsselwort `static`.

<sup>2</sup> Die Initialisierung, d.h. die erstmalige Zuweisung eines Werts an eine Variable, kann auch erst zu einem späteren Zeitpunkt erfolgen.

```
int c = 256;
```

In C wird das Ist-Gleich-Zeichen = als Zuweisungsoperator genutzt, der den Ausdruck auf der rechten Seite in die Variablen auf der linken Seite abspeichert.<sup>3</sup> Eine erneute Angabe des Datentyps einer Variablen würde beim Übersetzen sogar eine Fehlermeldung des Compilers zur Folge haben, da in diesem Fall von einer (versehentlichen) doppelten Vergabe eines Variablennamens ausgegangen wird.

Variablennamen dürfen in C maximal 31 Stellen lang sein. Sie können aus den Buchstaben A-Z und a-z, den Ziffern 0-9 und dem Unterstrich bestehen. Die einzige Einschränkung besteht darin, dass am Anfang von Variablennamen keine Ziffern stehen dürfen; Unterstriche am Anfang von Variablennamen sind zwar erlaubt, sollten aber vermieden werden, da diese üblicherweise für Bibliotheksfunktionen reserviert sind.

In C wird allgemein zwischen Groß- und Kleinschreibung unterschieden, beispielsweise bezeichnen a und A zwei unterschiedliche Variablen. Im Allgemeinen werden Variablen und Funktionen in C-Programmen fast immer klein geschrieben.

Ist einmal festgelegt, um welchen Datentyp es sich bei einer Variablen handelt, wird die Variable im Folgenden ohne Angabe des Datentyps verwendet.

## Elementare Datentypen

Als grundlegende Datentypen wird in C zwischen folgenden Arten unterschieden:

Typ	Bedeutung	Speicherbedarf
char	Ein einzelnes Zeichen	1 Byte (= 8 Bit)
int	Eine ganzzahlige Zahl	4 Byte (= 32 Bit)
short	Eine ganzzahlige Zahl	2 Byte (= 16 Bit)
long	Eine ganzzahlige Zahl	8 Byte (= 64 Bit)
float	Eine Fließkomma-Zahl	4 Byte (= 32 Bit)
double	Eine Fließkomma-Zahl	8 Byte (= 64 Bit)

Der Speicherbedarf der einzelnen Datentypen hängt von der konkreten Rechnerarchitektur ab; in der obigen Tabelle sind die Werte für 32-Bit-Systeme angegeben, die für Monocore-Prozessoren üblich sind. Auf anderen Systemen können sich andere Werte für die einzelnen Datentypen ergeben. Die Größe der Datentypen auf dem gerade verwendeten Rechner kann mittels des *sizeof*-Operators geprüft werden:

```
// Datei: sizeof.c

#include <stdio.h>

void main()
```

(continues on next page)

<sup>3</sup> Der Wertevergleich, wie er in der Mathematik durch das Ist-Gleich-Zeichen ausgedrückt wird, erfolgt in C durch den Operator ==.

```

printf("Size of char: %lu\n", sizeof (char));
printf("Size of int: %lu\n", sizeof (int));
printf("Size of short: %lu\n", sizeof (short));
printf("Size of long: %lu\n", sizeof (long));
printf("Size of float: %lu\n", sizeof (float));
printf("Size of double: %lu\n", sizeof (double));

```

}

In diesem Beispiel-Programm werden nach dem Compilieren mittels `gcc -o sizeof sizeof.c` und einem Aufruf von `./sizeof` die Größen der einzelnen Datentypen in Bytes ausgegeben. Hierzu wird bei der Funktion `printf()` das Umwandlungszeichen `%lu` verwendet, das durch den Rückgabewert von `sizeof` (entspricht `long integer`) ersetzt wird.

Einen „Booleschen“ Datentyp, der die Wahrheitswerte `True` oder `False` repräsentiert, existiert in C nicht. Stattdessen wird der Wert Null für `False` und jeder von Null verschiedene Wert als `True` interpretiert.

Komplexere Datentypen lassen sich aus diesen elementaren Datentypen durch Aneinanderreihungen (*Felder*) oder Definitionen von Strukturen (`struct`) erzeugen. Zusätzlich existiert in C ein Datentyp namens `void`, der null Bytes groß ist und beispielsweise dann genutzt wird, wenn eine Funktion *keinen* Wert als Rückgabe liefert.

## Modifier

Alle grundlegenden Datentypen (außer `void`) können zusätzlich mit einem der folgenden „Modifier“ versehen werden:

- `signed` bzw. `unsigned`:

Ohne explizite Angabe dieses Modifiers werden Variablen üblicherweise als `signed`, d.h. mit einem Vorzeichen versehen, interpretiert. Beispielsweise lassen sich durch eine 1 Byte (8 Bit) große Variable vom Typ `signed char` Werte von `-128` bis `+128` abbilden, durch eine Variable vom Typ `unsigned char` Werte von `0` bis `255`. Diese Werte werden dann üblicherweise als ASCII-Codes interpretiert.

- `extern`:

Dieser Modifier ist bei der Deklaration einer Variablen nötig, wenn diese bereits in einer anderen Quellcode-Datei definiert wurde. Für externe Variablen wird kein neuer Speicherplatz reserviert. Gleichzeitig wird durch den `extern`-Modifier dem Compiler mitgeteilt, in den zu Beginn eingebundenen Header-Dateien nach einer Variablen dieses Namens zu suchen und den dort reservierten Speicherplatz gemeinsam zu nutzen.

- `static`:

(Fortsetzung der vorherigen Seite)

```
// *b = 15;           // Fataler Fehler, Speicheradresse nicht bekannt!
// !!!
```

  

```
// Zeiger IMMER erst initialisieren:
b = &a;             // Der Zeiger zeigt jetzt auf die Adresse von a
*b = 15;            // Zuweisung in Ordnung!
```

Wäre der Zeiger auf der linken Seite gleich `NULL`, so würde die Wertzuweisung an eine undefinierte Stelle erfolgen; im schlimmsten Fall würde eine andere für das Programm wichtige Speicheradresse überschrieben werden. Ein solcher Fehler kann vom Compiler nicht erkannt werden, kann aber mit großer Wahrscheinlichkeit ein abnormales Verhalten des Programms oder einen Absturz zur Folge haben.

## Felder

Als Feld („Array“) bezeichnet man eine Zusammenfassung von mehreren Variablen gleichen Datentyps zu einem gemeinsamen Speicherbereich.

Bei der Definition eines Arrays muss einerseits der im Array zu speichernde Datentyp angegeben werden, andererseits wird zusätzlich in eckigen Klammern die Größe des Arrays angegeben. Damit ist festgelegt, wie viele Elemente in dem Array maximal gespeichert werden können.<sup>3</sup> Die Syntax lautet somit beispielsweise:

```
int numbers[10];
```

  

```
// Definition und Zuweisung zugleich:
int other_numbers[5] = { 10, 11, 12, 13, 14 };
```

Wird ein Array bei der Definition gleich mit einem konkreten Inhalt initialisiert, so kann die explizite Größenangabe entfallen und anstelle dessen ein leeres Klammerpaar `[]` gesetzt werden.

Der Hauptvorteil bei der Verwendung von Arrays liegt darin, eine Vielzahl gleichartiger Datei über eine einzige Variable (den Namen des Arrays) ansprechen zu können. Auf die einzelnen Elemente eines Feldes kann nach im eigentlichen Programm mittels des so genannten Selektionsoperators `[]` zugegriffen werden. Zwischen die eckigen Klammern wird dabei ein (ganzzahliger) Laufindex `i` geschrieben.

Hat ein Array insgesamt `n` Elemente, so kann der Laufindex `i` alle ganzzahligen Werte zwischen 0 und `n-1` annehmen. Das erste Element hat also den Index 0, das zweite den Index 1, das letzte schließlich den Index `n-1`. Somit kann der Inhalt jeder im Array gespeicherten Variablen ausgelesen oder durch einen anderen ersetzt werden:

<sup>3</sup> Die Größe von Feldern kann nach der Deklaration nicht mehr verändert werden. Somit muss das Feld ausreichend groß gewählt werden, um alle zu erwartenden Werte speichern zu können. Andererseits sollte es nicht unnötig groß gewählt werden, da ansonsten auch unnötig viel Arbeitsspeicher reserviert wird.

Soll die Größe eines Feldes erst zur Laufzeit festgelegt werden, so müssen die Funktionen `malloc()` bzw. `calloc()` verwendet werden.

```

int numbers[5];

numbers[0] = 3;
numbers[1] = 5;
numbers[2] = 8;
numbers[3] = 13;
numbers[4] = 21;

printf("Die vierte Nummer des Feldes 'num' ist %i.\n", numbers[3]);

```

Eine Besonderheit von Arrays in C ist es, dass der Compiler beim Übersetzen nicht prüft, ob bei der Verwendung eines Laufindex die Feldgrenzen eingehalten werden. Im Fall eines Arrays `numbers` mit fünf Elementen könnte beispielsweise mit `numbers[5] = 1` ein Eintrag in einen Speicherbereich geschrieben werden, der außerhalb des Arrays liegt. Auf korrekte Indizes muss somit der Programmierer achten, um Programmfehler zu vermeiden.

## Mehrdimensionale Felder

Ein Array kann wiederum Arrays als Elemente beinhalten. Beispielsweise kann man sich eine Tabelle aus einer Vielzahl von Zeilen zusammengesetzt denken, die ihrerseits wiederum eine Vielzahl von Spalten bestehen können. Beispielsweise könnte ein solches Tabellen-Array, das als Einträge jeweils Zahlen erwartet, folgendermaßen deklariert werden:<sup>4</sup>

```

// Tabelle mit 3 Zeilen und je 4 Spalten deklarieren:
int zahlentabelle[3][4];

```

Auch in diesem Fall laufen die Indexwerte bei  $n$  Einträgen nicht von 1 bis  $n$ , sondern von 0 bis  $n - 1$ . Der erste Auswahloperator greift ein Zeilenelement heraus, der zweite eine bestimmte Spalte der ausgewählten Zeile. Auch eine weitere Verschachtelung von Arrays nach dem gleichen Prinzip ist möglich, wobei der Zugriff auf die einzelnen Werte meist über `for`-Schleifen erfolgt.

## Zeiger auf Felder

In C sind Felder und Zeiger eng miteinander verwandt: Gibt man den Namen einer Array-Variablen ohne eckige Klammern an, so entspricht dies einem Zeiger auf die erste Speicheradresse, die vom Array belegt wird; nach der Deklaration `int numbers[10];` kann also beispielsweise als abkürzende Schreibweise für das erste Element des Feldes anstelle von `&numbers[0]` auch die Kurzform `numbers` benutzt werden.<sup>5</sup>

<sup>4</sup> Eine direkte Initialisierung eines mehrdimensionalen Arrays ist ebenfalls unmittelbar möglich; dabei werden die einzelnen „Zeilen“ für eine bessere Lesbarkeit in geschweifte Klammern gesetzt. Beispielsweise kann gleich bei der Definition `int zahlentabelle[3][4] = { {3,4,1,5}, {8,5,6,9}, {4,7,0,3} };` geschrieben werden.

<sup>5</sup> Legt man bei der Deklaration eines Feldes seine Groesse nicht fest, um diese erst zur Laufzeit mittels `malloc()` zu reservieren, so kann bei der Deklaration anstelle von `int numbers[];` ebenso `int *numbers;` geschrieben werden.

Da alle Elemente eines Arrays den gleichen Datentyp haben und somit gleich viel Speicherplatz belegen, unterscheiden sich die einzelnen Speicheradressen der Elemente um die Länge des Datentyps, beispielsweise um `sizeof (int)` für ein Array mit `int`-Werten oder `sizeof (float)` für ein Array mit `float`-Werten. Ausgehend vom ersten Element eines Arrays erhält man somit die weiteren Elemente des Feldes, indem man den Wert des Zeigers um das  $1, 2, \dots, n - 1$ -fache der Länge des Datentyps erhöht:

```
int numbers[10];
int *numpointer;

// Pointer auf erstes Element des Arrays:
numpointer = &numbers;                                // oder: &numbers[0]

// Pointer auf zweites Element des Arrays:
numpointer = &numbers + sizeof (int);                // oder: &numbers[1]

// Pointer auf drittes Element des Arrays:
numpointer = &numbers + 2 * sizeof (int);            // oder: &numbers[2]
```

Beim Durchlaufen eines Arrays ist eine Erhöhung des Zeigers in obiger Form auch mit dem *Inkrement-Operator* möglich: Es kann also auch `numpointer++` statt `numpointer = numpointer + sizeof (int)` geschrieben werden, um den Zeiger auf das jeweils nächste Element des Feldes zu bewegen; dies wird beispielsweise in `for`-Schleifen genutzt. Ebenso kann das Feld mittels `numpointer--` schrittweise rückwärts durchlaufen werden; auf das Einhalten der Feldgrenzen muss der Programmierer wiederum selbst achten.

Da es sich bei Speicheradressen um `unsigned int`-Werte handelt, können zwei Zeiger auch ihrer Größe nach verglichen werden. Hat man beispielsweise zwei Pointer `numpointer_1` und `numpointer_2`, die beide auf ein Element eines Arrays zeigen, so würde `numpointer_1 < numpointer_2` bedeuten, dass der erste Pointer auf ein Element zeigt, das sich weiter vorne im Array befindet. Ebenso kann in diesem Fall mittels `numpointer_2 - numpointer_1` die Anzahl der Elemente bestimmt werden, die zwischen den beiden Pointern liegen.

Anderer mathematische Operationen sollten auf Zeiger nicht angewendet werden; ebenso sollten Array-Variablen, obwohl sie letztlich einen Zeiger auf das erste Element des Feldes darstellen, niemals direkt inkrementiert oder dekrementiert werden, da das Array eine feste Stelle im Speicher einnimmt. Stattdessen definiert man stets einen Zeiger auf das erste Element des Feldes und inkrementiert diesen, um beispielsweise in einer Schleife auf die einzelnen Elemente eines Feldes zuzugreifen.

## Zeichenketten

Zeichenketten („Strings“), beispielsweise Worte und Sätze, stellen die wohl häufigste Form von Arrays dar. Eine Zeichenkette besteht aus einer Aneinanderreihung einzelner Zeichen (Datentyp `char`) und wird stets mit einer binären Null (`\0`) abgeschlossen. Beispielsweise entspricht die Zeichenkette "Hallo!" einem Array, das aus 'H', 'a', 'l', 'l', 'o', '!' und dem Zeichen `\0` besteht. Dieser Unterschied besteht allgemein zwischen Zei-

chenketten, die mit doppelten Hochkommata geschrieben werden, und einzelnen Zeichen, die in einfachen Hochkommata dargestellt werden.

Die Deklaration einer Zeichenkette entspricht der Deklaration eines gewöhnlichen Feldes:

```
// Deklaration ohne Initialisierung:  
char string_one[15];  
  
// Deklaration mit Initialisierung:  
char string_two[] = "Hallo Welt!"
```

Bei der Festlegung der maximalen Länge der Zeichenkette muss beachtet werden, dass neben den zu speichernden Zeichen auch Platz für das String-Ende-Zeichen '\0' bleiben muss. Als Programmierer muss man hierbei selbst darauf achten, dass die Feldgröße ausreichend groß gewählt wird.

Wird einer String-Variablen nicht bereits bei der Deklaration eine Zeichenkette zugewiesen, so ist dies anschliessend zeichenweise (beispielsweise mittels einer *Schleife*) möglich:

```
string_one[0] = 'H';  
string_one[1] = 'a';  
string_one[2] = 'l';  
string_one[3] = 'l';  
string_one[4] = 'o';  
string_one[5] = '!';  
string_one[6] = '\0';
```

Eine Zuweisung eines ganzen Strings an eine String-Variable in Form von `string_one = "Hallo!"` ist nicht direkt möglich, sondern muss über die Funktion `strcpy()` aus der Standard-Bibliothek `string.h` erfolgen:

```
// Am Dateianfang:  
#include <string.h>  
  
// ...  
  
// String-Variable deklarieren:  
char string_one[15];  
  
// Zeichenkette in String-Variable kopieren:  
strcpy(string_one, "Hallo Welt!");  
  
// Zeichenkette ausgeben:  
printf("%s\n", string_one);
```

Anstelle der Funktion `strcpy()` kann auch die Funktion `strncpy()` verwendet werden, die nach der zu kopierenden Zeichenkette noch einen `int`-Wert `n` erwartet; diese Funktion kopiert maximal `n` Zeichen in die Zielvariable, womit ein Überschreiten der Feldgrenzen ausgeschlossen werden kann.

## ASCII-Codes und Sonderzeichen

Die einzelnen Zeichen (Datentyp `char`) werden vom Computer intern ebenfalls als ganzzählige Werte ohne Vorzeichen behandelt. Am weitesten verbreitet ist die so genannte ASCII-Codierung („American Standard Code for Information Interchange“), deren Zuweisungen in der folgenden *ASCII-Tabelle* abgebildet sind. Wird beispielsweise nach der Deklarierung `char c;` der Variablen `c` mittels `c = 120` ein numerischer Wert zugewiesen, so liefert die Ausgabe von `printf("%c\n", c);` den zur Zahl 120 gehörenden ASCII-Code, also `x`.

Dez	AS-CII														
0	NUL	16	DLE	32	SP	48	0	64	©	80	P	96	‘	112	p
1	SOH	17	DC1	33	!	49	1	65	A	81	Q	97	a	113	q
2	STX	18	DC2	34	"	50	2	66	B	82	R	98	b	114	r
3	ETX	19	DC3	35	#	51	3	67	C	83	S	99	c	115	s
4	EOT	20	DC4	36	\$	52	4	68	D	84	T	100	d	116	t
5	ENQ	21	NAK	37	%	53	5	69	E	85	U	101	e	117	u
6	ACK	22	SYN	38	&	54	6	70	F	86	V	102	f	118	v
7	BEL	23	ETB	39	'	55	7	71	G	87	W	103	g	119	w
8	BS	24	CAN	40	(	56	8	72	H	88	X	104	h	120	x
9	HT	25	EM	41	)	57	9	73	I	89	Y	105	i	121	y
10	LF	26	SUB	42	*	58	:	74	J	90	Z	106	j	122	z
11	VT	27	ESC	43	+	59	;	75	K	91	[	107	k	123	{
12	FF	28	FS	44	,	60	<	76	L	92	\	108	l	124	
13	CR	29	GS	45	-	61	=	77	M	93	]	109	m	125	}
14	SO	30	RS	46	.	62	>	78	N	94	^	110	n	126	~
15	SI	31	US	47	/	63	?	79	O	95	-	111	o	127	DEL

Die zu den Zahlen 0 bis 127 gehörenden Zeichen sind bei fast allen Zeichensätzen identisch. Da der ASCII-Zeichensatz allerdings auf die englische Sprache ausgerichtet ist und damit keine Unterstützung für Zeichen anderer Sprachen beinhaltet, gibt es Erweiterungen des ASCII-Zeichensatzes für die jeweiligen Länder.

Neben den Obigen ASCII-Zeichen können Zeichenketten auch so genannte „Escape-Sequenzen“ als Sonderzeichen beinhalten. Der Name kommt daher, dass zur Darstellung dieser Zeichen ein Backslash-Zeichen \ erforderlich ist, das die eigentliche Bedeutung des darauf folgenden Zeichens aufhebt. Einige wichtige dieser Sonderzeichen sind in der folgenden Tabelle aufgelistet.

Zeichen	Bedeutung
\n	Zeilenwechsel („new line“)
\t	Tabulator (entspricht üblicherweise 4 Leerzeichen)
\b	Backspace
\\"	Backslash-Zeichen
\\""	Doppeltes Anführungszeichen
\'	Einfaches Anführungszeichen

Eine weitere Escape-Sequenz ist das Zeichen '\0' als Endmarkierung einer Zeichenkette, das verständlicherweise jedoch nicht innerhalb einer Zeichenketten stehen darf.

# Ausgabe und Eingabe

Das Ausgeben und Einlesen von Daten über den Bildschirm erfolgt häufig mittels der Funktionen `printf()` und `scanf()`.<sup>1</sup> Beide Funktionen sind Teil der *Standard-Bibliothek stdio.h*, so dass diese zu Beginn der Quellcode-Datei mittels `include <stdio.h>` eingebunden werden muss.<sup>2</sup>

## **printf() – Daten formatiert ausgeben**

Die Funktion `printf()` dient grundsätzlich zur direkten Ausgabe von Zeichenketten auf dem Bildschirm; beispielsweise gibt `printf("Hallo Welt!")` die angegebene Zeichenkette auf dem Bildschirm aus. Innerhalb der Zeichenketten können allerdings Sonderzeichen sowie Platzhalter für beliebige Variablen und Werte eingefügt werden.

Zeichen	Bedeutung
\n	Neue Zeile
\t	Tabulator (4 Leerzeichen)
\\\	Backslash-Zeichen \
\'	Einfaches Anführungszeichen
\\"	Doppeltes Anführungszeichen

Die in der obigen Tabelle angegebenen Sonderzeichen werden auch „Escape-Sequenzen“ genannt, da sie nur mittels des vorangehenden Backslash-Zeichens, das ihre sonstige Bedeutung aufhebt, innerhalb einer Zeichenkette dargestellt werden können.

Ein Platzhalter besteht aus einem %-Zeichen, gefolgt von einem oder mehreren Zeichen, welche den Typ der auszugebenden Werte oder Variablen angeben und gleichzeitig festlegen, wie die Ausgabe formatiert werden soll. Damit kann beispielsweise bestimmt werden, wie viele Stellen für einen Wert reserviert werden sollen, ob die Ausgabe links- oder rechtsbündig erfolgen soll, und/oder ob bei der Ausgabe von Zahlen gegebenenfalls führende Nullen angefügt werden sollen.

<sup>1</sup> Um Daten von Dateien anstelle vom Bildschirm einzulesen, gibt es weitere Funktionen, die im Abschnitt *Dateien und Verzeichnisse* näher beschrieben sind.

<sup>2</sup> Genau genommen erfolgt bei der Funktion `printf()` die Ausgabe auf den Standard-Ausgang (`stdout`). Bei diesem handelt es sich als Voreinstellung um den Bildschirm, in speziellen Fällen kann jedoch mittels der Funktion `fopen()` auch eine beliebige Datei oder ein angeschlossenes Gerät als Standard-Ausgang festgelegt werden.

Ebenso liest die Funktion `scanf()` vom Standard-Eingang (`stdin`) ein, der als Voreinstellung wiederum dem Bildschirm entspricht.

```

// Den Wert Pi auf sechs Nachkommastellen genau ausgeben:

printf("Der Wert von Pi ist %.6f...\n", 3.141592653589793)
// Ergebnis: Der Wert von Pi ist 3.141593...

// Maximal dreistellige Zahlen rechtsbündig ausgeben:

printf("%3i:\n%3i:\n%3i:\n", 1, 10, 100);
// Ergebnis:
// 1:
// 10:
// 100:

// Maximal dreistellige Zahlen linksbündig ausgeben:

printf("%3i:\n%3i:\n%3i:\n", 1, 10, 100);
// Ergebnis:
// 1 :
// 10 :
// 100:

// Einstelligen Zahlen eine Null voranstellen:

printf("%02i.: \n%02i.: \n%02i.: \n", 8, 9, 10);
// Ergebnis:
// 08.:
// 09.:
// 10.:

```

In den obigen Beispielen wurden der Funktion `printf()` zwei oder mehr Argumente übergeben. Beim ersten Argument handelt es sich um einen so genannten Formatstring, bei den folgenden Argumenten um die auf dem Bildschirm auszugebenden Werte. Falls diese, wie im ersten Beispiel, mehr Nachkommastellen haben als in der Formatierung vorgesehen (Die Angabe `%.6f` steht für einen Wert vom Datentyp `float` sechs Nachkommastellen), so wird der Wert automatisch auf die angegebene Genauigkeit gerundet.

Zur Festlegung des Datentyps einer auszugebenden Variablen gibt es allgemein folgende Umwandlungszeichen:

Zeichen	Argument	Bedeutung
d, i	int	Dezimal-Zahl mit Vorzeichen.
o	int	Oktal-Zahl ohne Vorzeichen (und ohne führende Null).
x, X	int	Hexadezimal-Zahl ohne Vorzeichen (und ohne führendes 0x oder 0X), also abcdef bei 0x oder ABCDEF bei 0X.
u	int	Dezimal-Zahl ohne Vorzeichen.
c	int	Ein einzelnes Zeichen ( <code>unsigned char</code> ).
s	char *	Zeichen einer Zeichenkette bis zum Zeichen \0, oder bis zur angegebenen Genauigkeit.
f	double	Dezimal-Zahl als [-]mmm.ddd, wobei die angegebene Genauigkeit die Anzahl der d festlegt. Die Voreinstellung ist 6, bei 0 entfällt der Dezimalpunkt.
e, E	double	Dezimal-Zahl als [-]m.ddddde±xx oder [-]m.dddddE±xx, wobei die angegebene Genauigkeit die Anzahl der d festlegt. Die Voreinstellung ist 6, bei 0 entfällt der Dezimalpunkt.
g, G	double	Dezimal-Zahl wie wie %e oder %E. Wird verwendet, wenn der Exponent kleiner als die angegebene Genauigkeit ist; unnötige Nullen am Schluss werden nicht ausgegeben.
p	void *	Zeiger (Darstellung hängt von Implementierung ab).
n	int *	Anzahl der aktuell von printf() ausgegebenen Zeichen.

Die obigen Formatangaben lassen sich durch Steuerzeichen („flags“) zwischen dem %- und dem Umwandlungszeichen weiter modifizieren:

- **Zahl:** Minimale Feldbreite festlegen: Das umgewandelte Argument wird in einem Feld ausgegeben, das mindestens so breit ist, bei Bedarf aber auch breiter. Hat das umgewandelte Argument weniger Zeichen als die Feldbreite es verlangt, so werden auf der linken Seite Leerzeichen eingefügt.
- **.Zahl:** Genauigkeit von Gleitkommazahlen festlegen: Gibt die maximale Anzahl von Zeichen an, die nach dem Dezimalpunkt ausgegeben werden
- **-:** Ausrichten des umgewandelten Arguments am linken Rand des Ausgabefeldes (Leerzeichen werden bei Bedarf nicht links, sondern rechts eingefügt)
- **+:** Ausgabe einer Zahl stets mit Vorzeichen
- **Leerzeichen:** Ausgabe eines Leerzeichens vor einer Zahl, falls das erste Zeichen kein Vorzeichen ist
- **0:** Zahlen bei der Umwandlungen bis zur Feldbreite mit führenden Nullen aufüllen

Anstelle einer Zahl kann auch das Zeichen \* als Feldbreite angegeben werden. In diesem Fall wird die Feldbreite durch eine zusätzlich an dieser Stelle in der Argumentliste angegebenen int-Variablen festgelegt:

```
int zahl = 1000;
int breite = 5;

printf("Der Wert von der Variable \"zahl\" ist: %*d", breite, zahl);
```

Die Formatangaben %e und %g können gleichermaßen zur Ausgabe von Gleitkommazahlen in der Zehnerpotenz-Schreibweise verwendet werden. Sie unterscheiden sich nur bei Zahlen mit wenig Nachkommastellen. Beispielsweise würde die Ausgabe `printf("%g\n", 2.15);` als Ergebnis 2.15 anzeigen, während `printf("%e\n", 2.15);` als Ergebnis 2.150000e+00 liefern würde.

Soll eine `long`-Variante eines Integers ausgegeben werden, so muss vor das jeweilige Umwandlungszeichen ein `l` geschrieben werden, beispielsweise `lu` für `long unsigned int` oder `ld` für `long int`; für `long double` wird `L` geschrieben.

Soll das `%`-Zeichen innerhalb einer Zeichenkette selbst ausgegeben werden, so muss an dieser Stelle `%%` geschrieben werden.

Soll über mehrere Zeilen hinweg Text mittels `printf()` ausgegeben werden, so ist meist es für eine bessere Lesbarkeit empfehlenswert, für jede neue Zeile eine eigene `printf()`-Anweisung zu schreiben.

## puts() – Einzelne Zeichenketten ausgeben

Sollen nur einfache Zeichenketten (ohne Formatierung und ohne Variablenwerte) ausgegeben werden, so kann anstelle von `printf()` auch die Funktion `puts()` aus der Standard-Bibliothek `stdio.h` verwendet werden. Die in der Tabelle *Escape-Sequenzen* aufgelisteten Sonderzeichen können auch bei `puts()` verwendet werden, es muss jedoch am Ende einer Ausgabezeile kein `\n` angehängt werden; `puts()` gibt automatisch jeden String in einer neuen Zeile aus.

## putchar() – Einzelne Zeichen ausgeben

Mittels `putchar()` können einzelne Zeichen auf dem Bildschirm ausgegeben werden. Diese Funktion wird nicht nur von den anderen Ausgabefunktionen aufgerufen, sondern kann auch verwendet werden, wenn beispielsweise eine Datei zeichenweise eingelesen und nach Anwendung eines Filters wieder zeichenweise auf dem Bildschirm ausgegeben werden soll.<sup>3</sup>

## scanf() – Daten formatiert einlesen

Die Funktion `scanf()` kann als flexible Funktion verwendet werden, um Daten direkt vom Bildschirm beziehungsweise von der Tastatur einzulesen. Dabei wird bei `scanf()`, ebenso wie bei `printf()`, ein Formatstring angegeben, der das Format der Eingabe festlegt. Die Funktion weist dann die eingelesenen Daten, die dem Format entsprechen, vom Bildschirm ein und weist ihnen eine oder mehrere Programmvariablen zu. Im Formatstring können die gleichen *Umwandlungszeichen* wie bei `printf()` verwendet werden.

Die Eingabe mittels `scanf()` erfolgt „gepuffert“, d.h. die mit der Tastatur eingegebenen Zeichen werden zunächst in einem Zwischenspeicher („Puffer“) des Betriebssystems abgelegt. Erst, wenn der Benutzer die Enter-Taste drückt, wird der eingegebene Text von `scanf()` verarbeitet.

<sup>3</sup> Streng genommen handelt es sich bei `putchar()` nicht um eine Funktion, sondern um ein *Makro*: Letztlich wird `putchar(Zeichen)` vom Präprozessor durch einen Funktionsaufruf von `fputc(Zeichen, stdin)` ersetzt. Die Funktion `fputc()` wird im Abschnitt *Dateien und Verzeichnisse* näher beschrieben.

Bei der Zuweisung der eingelesenen Daten wird bei Benutzung der Funktion `scanf()` nicht der jeweilige Variablenname, sondern stets die zugehörige Speicheradresse angegeben, an welcher die Daten abgelegt werden sollen; diese kann leicht mittels des *Adress-Operators* `&` bestimmt werden. Um also beispielsweise einen `int`-Wert vom Bildschirm einzulesen, gibt man folgendes ein:

```
int n;

// Benutzer zur Eingabe auffordern:
printf("Bitte einen ganzzahligen Wert eingeben: ")

// Eingegebenen Wert einlesen:
scanf("%i", &n);
```

Sobald der Benutzer seine Eingabe mit `Enter` bestätigt, wird im obigen Beispiel die eingegebene Zahl eingelesen und am Speicherplatz der Variablen `n` hinterlegt.

Zum Einlesen von Zeichenketten muss dem Variablenamen kein `&` vorangestellt werden, da es sich bei einer Zeichenkette um ein *Array* handelt. Dieses wiederum entspricht einem *Zeiger* auf den ersten Eintrag, und ab eben dieser Stelle soll die eingelesene Zeichenkette abgelegt werden. Beim Einlesen von Daten in Felder muss allerdings beachtet werden, dass der angegebene Zeiger bereits *initialisiert* wurde. Eine simple Methode, um dies sicherzustellen, ist dass eine String-Variablen nicht mit `char *mystring;`, sondern beispielsweise mit `char mystring[100];` definiert wird.

## Whitespace als Trennzeichen

Mit einer einzelnen `scanf()`-Funktion können auch mehrere Werte gleichzeitig eingelesen werden, wenn mehrere Umwandlungszeichen im Formatstring und entsprechend viele Speicheradressen als weitere Argumente angegeben werden. Beim Einlesen achtet `scanf()` dabei so genannte Whitespace-Zeichen (Leerzeichen, Tabulator-Zeichen oder Neues-Zeile-Zeichen), um die einzelnen Daten voneinander zu trennen. Soll der Benutzer beispielsweise zwei beliebige Zahlen eingeben, so können diese mit einem einfachen Leerzeichen zwischen ihnen, aber ebenso in zwei getrennten Zeilen eingegeben werden.

```
int n1, n2;

// Benutzer zur Eingabe auffordern:
printf("Bitte zwei beliebige Werte eingeben: ")

// Eingegebene Werte einlesen:
scanf("%f %f", &n1, &n2);
```

## `fflush()` – Zwischenspeicher löschen

Da die Daten bei Verwendung von `scanf()` zunächst in einen Zwischenspeicher eingelesen werden, können Probleme auftreten, wenn der Benutzer mehr durch Whitespace-Zeichen getrennte Werte eingibt, als beim Aufruf der Funktion `scanf()` verarbeitet werden. Die

restlichen Werte verbleiben in diesem Fall im Zwischenspeicher und würden beim nächsten Aufruf von `scanf()` noch vor der eigentlich erwarteten Eingabe verarbeitet werden. Eine Abhilfe hierfür schafft die Funktion `fflush()`, die nach jedem Aufruf von `scanf()` aufgerufen werden sollte und ein Löschen aller noch im Zwischenspeicher abgelegten Werte bewirkt.

Beim Einlesen von Zeichenketten mittels `%s` ist das wortweise Einlesen von `scanf()` oftmals hinderlich, da in der mit `%s` verknüpften Variable nur Text bis zum ersten Whitespace-Zeichen (Leerzeichen, Tabulator-Zeichen oder Neues-Zeile-Zeichen) gespeichert wird. Ganze Zeilen, die aus beliebig vielen Wörtern bestehen, sollten daher bevorzugt mittels `gets()` oder `fgets()` eingelesen werden.

## gets() und fgets() – Einzelne Zeichenketten einlesen

Um eine Textzeile auf einmal einzulesen, kann die Funktion `gets()` aus der Standard-Bibliothek `stdio.h` verwendet werden. Diese Funktion liest eine Textzeile vom Bildschirm ein und speichert sie in der angegebenen Variablen ein:

```
int mystring[81];  
  
gets(mystring);
```

Ein Neues-Zeile-Zeichen `\n` am Ende des Eingabestrings wird von `gets()` automatisch abgeschnitten, das Zeichen `\0` zum Beenden der Zeichenkette automatisch angefügt. Wichtig ist allerdings bei der Verwendung von `gets()`, dass der angegebene String-Pointer auf ein ausreichend großes Feld zeigt. Im obigen Beispiel darf die eingelesene Zeile somit nicht mehr als 80 Zeichen haben, da auch noch Platz für das Zeichen `\0` bleiben muss. Werden die Feldgrenzen überschritten, kann dies ein unkontrolliertes Verhalten des Programms oder gar einen Programmabsturz zur Folge haben.<sup>4</sup>

Als bessere Alternative zu `gets()` kann die Funktion `fgets()` verwendet werden, welche die Anzahl der maximal eingelesenen Zeichen beschränkt:

```
int mystring[81];  
int n = 80;  
  
fgets(mystring, n, stdin);
```

Im Unterschied zu `gets()` speichert `fgets()` das Neue-Zeile-Zeichen `\n` mit in der eingelesenen Zeichenkette, was unter Umständen bei der Längenangabe `n` berücksichtigt werden muss. Die Funktion `fgets()` gibt, wenn eine Zeichenkette erfolgreich eingelesen wurde, einen Zeiger als Ergebnis zurück, der mit der Speicheradresse der angegebenen Stringvariablen übereinstimmt; bei einem Fehler wird `NULL` als Ergebnis zurück gegeben. Um eine Textzeile auf einmal einzulesen, kann die Funktion `gets()` aus der Standard-Bibliothek

<sup>4</sup> Im neuen C11-Standard wird `gets()` aufgrund seiner Fehleranfälligkeit nicht mehr als Standard gelistet, den ein Compiler abdecken muss. Da die Funktion in sehr vielen Programmcodes vorkommt, wird `gcc` wohl auch in absehbarer Zukunft diese Funktion unterstützen. In C11 wurde dafür die ähnliche Funktion `gets_s()` im optionalen Teil von `stdio.h` aufgenommen, die jedoch ebenfalls nicht jeder Compiler zwingend unterstützen muss. Dies ist ein weiterer Grund, bevorzugt `fgets()` zu verwenden.

$4 \% 5 / 2$  alle Operatoren die gleiche Priorität, sie werden gemäß ihrer Assoziativität von links nach rechts ausgewertet, so dass der Ausdruck formal mit  $((3 * 4) \% 5) / 2$  identisch ist; somit ist das Ergebnis gleich  $(12 \% 5) / 2 = 2 / 2 = 1$ .

Zur besseren Lesbarkeit können Teil-Aussagen die durch einen Operator mit höherer Priorität verbunden sind jederzeit, auch wenn es nicht notwendig ist, in runde Klammern gesetzt werden, ohne den Wert der Aussage zu verändern.

## Funktionen

Funktionen werden verwendet, um einzelne, durch geschweifte Klammern begrenzte Code-Blöcke mit einem Namen zu versehen. Damit können Funktionen an beliebigen anderen Stellen im Programm aufgerufen werden.

Eine Funktion kann somit als „Unterprogramm“ angesehen werden, dem gegebenenfalls ein oder auch mehrere Werte als so genannte „Argumente“ übergeben werden können und das je nach Definition einen Wert als Ergebnis zurück gibt.

Die Definition einer Funktion hat folgenden Aufbau:

```
// Definition einer Funktion:  
rueckgabe_typ funktionsname( arg1, arg2, ... )  
{  
    Anweisungen  
}
```

Der Rückgabe-Typ gibt den Datentyp an, den die Funktion zurück gibt, beispielsweise `int` für ein ganzzahliges Ergebnis oder `char *` für eine Zeichenkette. Liefert die Funktion keinen Wert zurück, wird `void` als Rückgabe-Typ geschrieben. Die Argumentenliste der Funktion kann entweder leer sein oder eine beliebige Anzahl an zu übergebenen Argumenten beinhalten, wobei jedes Argument aus einem Argument-Typ und einem Argument-Namen besteht. Beim Aufruf der Funktion müssen die Datentypen der übergebenen Werte mit denen der bei der Deklaration angegebenen Argumentliste übereinstimmen.<sup>1</sup>

Bezüglich der Anweisungen innerhalb eines Funktionsblocks bestehen kaum Einschränkungen, außer dass es nicht möglich ist, innerhalb einer Funktion weitere Funktionen zu definieren. Neue Variablen, deren Gültigkeit auf die jeweilige Funktion beschränkt ist, müssen stets zu Beginn des Funktionsblocks definiert werden. Am Ende der Funktion verlieren diese „lokalen“ Variablen standardmäßig wieder ihre Gültigkeit; soll eine Variable ihren Wert jedoch bis zum nächsten Aufruf der Funktion behalten, muss bei der Definition der Variablen das Schlüsselwort `static` verwendet werden.

Soll eine Funktion einen Wert als Ergebnis zurückzugeben, so muss innerhalb der Funktion das Schlüsselwort `return` gesetzt werden, gefolgt von einem C-Ausdruck. Wenn die Funktion an einer `return`-Anweisung ankommt, wird der Ausdruck ausgewertet und das Ergebnis an die aufrufende Stelle im Programm zurück gegeben. Zu beachten ist lediglich,

<sup>1</sup> Streng genommen werden die Argumente bei der Definition als „formale Parameter“ bezeichnet, die beim Aufruf übergebenen Werte hingegen werden „aktuelle Parameter“ oder schlicht Argumente genannt.

dass der von `return` zurück gelieferte Wert mit dem in der Funktionsdefinition angegebenen Datentyp übereinstimmt, damit der Compiler keine Fehlermeldung ausgibt.

Nach der Definition der Funktion kann diese an beliebigen Stellen im Code genutzt werden, sie kann also auch von anderen Funktionen aufgerufen werden. Um eine Funktion allerdings bereits aufrufen zu können, wenn ihre Definition erst an einer späteren Stelle der Datei erfolgt, muss am Dateianfang – wie bei Variablen – zunächst der Prototyp der Funktion deklariert werden:<sup>2</sup>

```
// Deklaration des Funktions-Prototyps:  
rueckgabe_typ funktionsname( arg1, arg2, ... );
```

Bei C-Programmen, die nur aus einer einzigen Datei bestehen, werden die Funktionsprototypen üblicherweise gemeinsam mit der Deklaration von Variablen an den Anfang der Datei geschrieben. Die konkrete Definition der Funktionen erfolgt dann üblicherweise nach der Definition der Funktion `main()`.

Um eine Funktion aufzurufen, wird der Name der Funktion in Kombination mit einer Argumentliste in runden Klammern angegeben:

```
// Aufruf einer Funktion:  
funktionsname( arg1, arg2, ... );
```

Beim Aufruf einer Funktion müssen die Anzahl der übergebenen Argumente und ihre Datentypen mit der Funktions-Definition übereinstimmen.

C-Programme bestehen letztlich aus einer Vielzahl an Funktionen, die jeweils möglichst eine einzige, klar definierte Teilaufgabe übernehmen; entsprechend sollte der Funktionsname auf den Zweck der Funktion hinweisen. Eine Funktion Funktion sollte ebenfalls nicht allzu umfangreich sein, nur wenige Funktionen bestehen aus mehr als 30 Zeilen Code.<sup>3</sup> Auf diese Weise lassen sich einerseits einzelne Code-Teile leichter wieder verwerten, andererseits kann dadurch beim Suchen nach Fehlern der zu hinterfragende Code-Bereich schneller eingegrenzt werden.

## Call by Value und Call by Reference

In C werden alle Argumente standardmäßig „by Value“ übergeben, das heißt, dass die übergebenen Werte beim Funktionsaufruf kopiert werden, und innerhalb der Funktion mit lokalen Kopien der Werte gearbeitet wird. Eine Funktion kann hierbei die Originalvariable nicht verändern.

Wenn eine Funktion übergebene Variablen jedoch verändern soll, so müssen anstelle der Variablenwerte die Adressen der jeweiligen Variablen übergeben werden. Eine derartige Übergabe wird als „Call by Reference“ bezeichnet: Anstelle der Variablen wird ein *Zeiger* auf die Variable als Argument übergeben. Ändert die Funktion den Wert der Speicher-

<sup>2</sup> Deklarationen von Funktionen sind für das Compilieren des Programms unerlässlich, da für jeden Funktionsaufruf geprüft wird, ob die Art und Anzahl der übergebenen Argumente korrekt ist.

<sup>3</sup> Eine Funktion sollte maximal 100 Zeilen umfassen. Die Hauptfunktion `main()` sollte nur Unterfunktionen aufrufen, um möglichst übersichtlich zu sein.

# Kontrollstrukturen

Im folgenden Abschnitt werden die grundlegenden Kontrollstrukturen vorgestellt, mit denen sich der Ablauf eines C-Programms steuern lässt.

## if, elif und else – Bedingte Anweisungen

Mit Hilfe des Schlüsselworts `if` kann an einer beliebigen Stelle im Programm eine Bedingung formuliert werden, so dass die Anweisung(en) im unmittelbar folgenden Code-Block nur dann ausgeführt werden, sofern die Bedingung einen wahren Wert (ungleich Null) ergibt.

Eine `if`-Anweisung ist also folgendermaßen aufgebaut:

```
if (Bedingung)
{
    Anweisungen
}
```

In den runden Klammern können mittels der logischen Verknüpfungsoperatoren `and` beziehungsweise `or` mehrere Teilbedingungen zu einer einzigen Bedingung zusammengefügt werden. Bei einer einzeiligen Anweisung können die geschweiften Klammern weggelassen werden. Liefert die Bedingung den Wert Null, so wird der Anweisungsblock übersprungen und das Programm fortgesetzt.

Eine `if`-Anweisung kann um den Zusatz `else` erweitert werden. Diese Konstruktion wird immer dann verwendet, wenn man zwischen *genau* zwei Alternativen auswählen möchte.

```
if (Bedingung)
{
    Anweisungen
}
else
{
    Anweisungen
}
```

Der Vorteil einer `if-else`-Bedingung gegenüber der Verwendung zweier `if`-Anweisungen besteht darin, dass nur einmalig eine Bedingung getestet wird und das Programm somit schneller ausgeführt werden kann.

Soll neben der `if`-Bedingung eine (oder mehrere) weitere Bedingung getestet werden, so kann dies mittels des kombinierten Schlüsselworts `else if` geschehen. Die `else if`-Anweisungen werden nur dann ausgeführt, wenn die `if`-Bedingung falsch und die `elif`-Bedingung wahr ist.

```
if (Bedingung_1)
{
    Anweisungen
}
else if (Bedingung_2)
{
    Anweisungen
}
```

Allgemein können in einer `if`-Struktur mehrere `else if`-Bedingungen, aber nur ein `else`-Block vorkommen.

## switch – Fallunterscheidungen

Mittels des Schlüsselworts `switch` kann in C eine Fallunterscheidung eingeleitet werden. Hierbei wird der nach dem Schlüsselwort `switch` in runden Klammern angegebene Ausdruck ausgewertet, und in Abhängigkeit des sich ergebenden Werts einer der folgenden Fälle ausgewählt:

```
switch (Ausdruck)
{
    case const_1:
        Anweisungen_1

    case const_2:
        Anweisungen_2

    ...
    default:
        Default-Anweisungen
}
```

Bei den Konstanten, mit denen der Wert von `Ausdruck` verglichen wird, muss es sich um `int`- oder `char`-Werte handeln, die nicht mehrfach vergeben werden dürfen. Trifft kein `case` zu, so werden die unter `default` angegebenen Anweisungen ausgeführt.

Trifft ein `case` zu, so werden die angegebenen Anweisungen ausgeführt, anschließend wird der Ausdruck mit den übrigen `case`-Konstanten verglichen. Möchte man dies vermeiden, so kann man am Ende der `case`-Anweisungen die Anweisung `break;` einfügen, die einen Abbruch der Fallunterscheidung an dieser Stelle zur Folge hat.

In C ist es auch möglich Anweisungen für mehrere `case`-Werte zu definieren. Die Syntax dazu lautet:

```

switch (Ausdruck)
{
    case const_1:
    case const_2:
    case const_3:
        Anweisungen
    ...
}

```

In diesem Fall werden die bei `case const_3` angegebenen Anweisungen auch aufgerufen, wenn die Vergleiche `case const_1` oder `case const_2` zutreffen.

## for und while – Schleifen

Eine `for`-Schleife ist folgendermaßen aufgebaut:

```

for ( Initialisierung; Bedingung; Inkrementierung )
{
    Anweisungen
}

```

Gelangt das Programm zu einer `for`-Schleife, so werden nacheinander folgende Schritte ausgeführt:

- Zunächst wird der Initialisierungs-Ausdruck ausgewertet. Dieser ist üblicherweise eine Zuweisung, die eine Zählvariable auf einen bestimmten Wert setzt.
- Als nächstes wird der Bedingungs-Ausdruck ausgewertet. Dieser ist normalerweise ein relationaler Ausdruck (Vergleich).

Wenn die Bedingung falsch ist, so wird die `for`-Schleife beendet, und das Programm springt zur nächsten Anweisung außerhalb der Schleife.

Wenn die Bedingung wahr ist, so werden die im folgenden Block angegebenen Anweisung(en) ausgeführt.

- Nach der Ausführung der Anweisungen wird der Inkrementierungs-Ausdruck ausgewertet; hierbei wird beispielsweise die Zählvariable oder der Index eines Arrays mit jedem Schleifendurchlauf um 1 erhöht. Anschließend wird wiederum der Bedingungs-Ausdruck geprüft und gegebenenfalls die Ausführung der Schleifenanweisungen fortgesetzt.

Innerhalb einer `for`-Anweisung können weitere `for`-Anweisungen auftreten, so dass auch über mehrere Zählvariablen iteriert werden kann. Bei einer nur einzeiligen Anweisung können die geschweiften Klammern weggelassen werden.

Soll eine Schleife vorzeitig beendet werden, so kann dies mittels des Schlüsselworts `break` erreicht werden: Trifft das Programm auf diese Anweisung, so wird die Schleife unmittelbar beendet. [#] Möchte man die Schleife nicht beenden, sondern nur den aktuellen Schleifendurchgang überspringen, so kann man das Schlüsselwort `continue` verwenden.

Trifft das Programm auf diese Anweisung, so wird der aktuelle Schleifendurchgang beendet, und das Programm fährt mit dem nächsten Schleifendurchgang fort.

Üblicherweise werden **for**-Schleifen verwendet, um mittels der Zählvariablen für eine bestimmte Anzahl von Durchläufen zu sorgen. Ist zu Beginn der Schleife nicht bekannt, wie häufig der folgende Anweisungsblock durchlaufen werden soll, wird hingegen meist eine **while**-Schleife eingesetzt.

Eine **while**-Schleife ist folgendermaßen aufgebaut:

```
while ( Bedingung )
{
    Anweisungen
}
```

Eine **while**-Schleife führt einen Anweisungsblock aus, solange die angegebene Bedingung wahr (nicht Null) ist. Das Programm wertet dabei zunächst den als Bedingung angegebenen Ausdruck aus, und nur falls dieser einen von Null verschiedenen Wert liefert, wird der Anweisungsblock ausgeführt. Ergibt der als Bedingung angegebene Ausdruck bereits bei der ersten Auswertung den Wert Null, so wird die **while**-Schleife übersprungen, ohne dass der Anweisungsblock ausgeführt wird.

Häufig werden **while**-Schleifen als Endlos-Schleifen verwendet, die einen (zunächst) wahren Ausdruck als Bedingung verwenden. Unter einer bestimmten Voraussetzung wird dann mittels einer **if**-Anweisung innerhalb des Schleifenblocks entweder der Bedingungsausdruck auf den Wert Null gesetzt oder die Schleife mittels **break** beendet.

Soll eine gewöhnliche **while**-Schleife, unabhängig von ihrer Bedingung, mindestens einmal ausgeführt werden, so wird in selteneren Fällen eine **do-while**-Schleife eingesetzt. Eines solche Schleife ist folgendermaßen aufgebaut:

```
do
{
    Anweisungen
} while ( Bedingung )
```

Da es stets möglich ist, eine **do-while**-Schleife auch mittels einer **while**-Schleife zu schreiben, werden letztere wegen ihrer besseren Lesbarkeit meist bevorzugt.

# Präprozessor, Compiler und Linker

Ein klassischer C-Compiler besteht aus drei Teilen: Einem Präprozessor, dem eigentlichen Compiler, und einem Linker:

- Der Präprozessor bereitet einerseits den Quellcode vor (entfernt beispielsweise Kommentare und Leerzeilen); andererseits kann er mittels der im nächsten Abschnitt näher beschriebenen Präprozessor-Anweisungen Ersetzungen im Quellcode vornehmen.
- Der Compiler analysiert den Quellcode auf lexikalische oder syntaktische Fehler, nimmt gegebenenfalls Optimierungen vor und wandelt schließlich die aufbereiteten Quellcode-Dateien in binäre Objekt-Dateien (Endung: .o) um.
- Der Linker ergänzt die Objekt-Dateien um verwendete Bibliotheken und setzt die einzelnen Komponenten zu einem ausführbaren Gesamt-Programm zusammen.

## Präprozessor-Anweisungen

Der Präprozessor lässt sich im Wesentlichen durch zwei Anweisungen steuern, die jeweils durch ein Hash-Symbol # zu Beginn der Anweisung gekennzeichnet sind und ohne einen Strichpunkt abgeschlossen werden:

### #include – Einbinden von Header-Dateien

Mittels #include können weitere Quellcode-Teile in das Programm integriert werden. Diese Dateien werden vom Präprozessor eingelesen und an Stelle der #include-Anweisung in die Datei geschrieben.

Unterschieden wird bei #include-Anweisungen zwischen Bibliotheken, die sich in einem Standardpfad im System befinden und dem Compiler bekannt sind, und lokalen *Header-Dateien*, die sich üblicherweise im gleichen Verzeichnis befinden. Die Bibliotheken aus dem Standard-Pfad erhalten um ihren Namen eckige Klammern, die Namen der lokalen Header-Dateien werden in doppelte Anführungszeichen gesetzt:

```
// Standard-Bibliothek stdio.h importieren:  
#include <stdio.h>  
  
// Lokale Header-Datei input.h importieren:  
#include "input.h"
```

## #define – Definition von Konstanten und Makros

Mittels `#define` können Konstanten oder Makros definiert werden. Bei der Definition einer Konstanten wird zunächst der zu ersetzende Name anschließend der zugehörige Wert angegeben:

```
# define HALLO "Hallo Welt!"  
# define PI 3.1415
```

Eine Großschreibung der Konstantennamen ist nicht zwingend nötig, ist in der Praxis jedoch zum Standard geworden, um Konstanten- von Variablennamen unterscheiden zu können. Nicht verwendet werden dürfen allerdings folgende Konstanten, die im Präprozessor bereits vordefiniert sind:

- `__LINE__`: Ganzzahl-Wert der aktuellen Zeilennummer
- `__FILE__`: Zeichenkette mit dem Namen der kompilierten Datei
- `__DATE__`: Zeichenkette mit aktuellem Datum
- `__TIME__`: Zeichenkette mit aktueller Uhrzeit

Eine Festlegung mittels `#define` bleibt allgemein bis zum Ende der Quelldatei bestehen. Soll eine erneute Definition einer Konstanten `NAME` erfolgen, so muss die bestehende Definition erst mittels `#undef NAME` rückgängig gemacht werden.

Bei der Definition eines Makros mittels `#define` wird zunächst der Name des Makros angegeben. In runden Klammern stehen dann, wie bei der Definition einer *Funktion*, die Argumente, die das Makro beim Aufruf erwartet.<sup>1</sup> Unmittelbar anschließend wird der Code angegeben, den das Makro ausführen soll.

```
# define QUADRAT(x) ((x)*(x))
```

Bei der Definition von Makros muss beachtet werden, dass der Präprozessor die Ersetzungen nicht wie ein Taschenrechner oder Interpreter, sondern wie ein klassischer Text-Editor vornimmt. Steht im Quellcode beispielsweise die Zeile `result = QUADRAT(n)`, so wird diese durch den Präprozessor gemäß dem obigen Makro zu `result = ((n)*(n))` erweitert. In diesem Fall erscheinen die Klammern als unnötig. Steht allerdings im Quellcode die Zeile `result = QUADRAT(n+1)`, so wird diese mit Hilfe der Klammern zu `((n+1)*(n+1))` erweitert. Ohne die zusätzlichen Klammern in der Makro-Definition würde der Ausdruck zu `n+1*n+1` erweitert werden, was ein falsches Ergebnis liefern würde.

Innerhalb von Makro-Definitionen kann ein spezieller Operatoren verwendet werden: Der Operator `#` kann auf einen Argumentnamen angewendet werden und setzt den Namen der konkret angegebenen Variablen in doppelte Anführungszeichen:<sup>2</sup>

<sup>1</sup> Zu beachten ist, dass bei der Definition eines Makros kein Leerzeichen zwischen dem Makronamen und der öffnenden runden Klammer der Argumentenliste vorkommen darf. Der Präprozessor würde ansonsten den Makronamen als Namen einer Konstanten interpretieren und den gesamten Rest der Zeile als Wert dieser Konstanten interpretieren.

<sup>2</sup> Zudem können mit dem zweiten möglichen Makro-Operator `##` die Namen von zwei oder mehreren übergebenen Argumenten zu einer neuen Bezeichnung verbunden werden. Dieser Operator wird allerdings nur sehr selten eingesetzt.

Gibt einen Zeiger auf erstes Vorkommen von der Zeichenkette `str_2` innerhalb der Zeichenkette `str_1` als Ergebnis zurück, oder `NULL`, falls diese nicht vorkommt.

- `size_t strlen(const char *str)`

Gibt die Länge der Zeichenkette `str` ohne `\0` an.

- `char * strerror(size_t n)`

Gibt einen Zeiger auf diejenige Zeichenkette als Ergebnis zurück, die dem Fehler mit der Nummer `n` zugewiesen ist.

- `char * strtok(char *str_1, const char *str_2)`

Durchsucht die Zeichenkette `str_1` nach Zeichenfolgen, die durch Zeichen aus der Zeichenkette `str_2` begrenzt sind.

## stdio.h – Ein- und Ausgabe

Die Datei `stdio.h` definiert Typen und Funktionen zum Umgang mit Datenströmen („Streams“). Ein Stream ist Quelle oder Ziel von Daten und wird mit einer Datei oder einem angeschlossenen Gerät verknüpft.

Unter Windows muss zwischen Streams für binäre und für Textdateien unterschieden werden, unter Linux nicht. Ein Textstream ist eine Folge von Zeilen, die jeweils kein oder mehrere Zeichen enthalten und jeweils mit `'\n'` abgeschlossen sind.

Ein Stream wird mittels der Funktion `open()` mit einer Datei oder einem Gerät verbunden; die Verbindung wird mittels der Funktion `close()` wieder aufgehoben. Öffnet man eine Datei, so erhält man einen Zeiger auf ein Objekt vom Typ `FILE`, in welchem alle Information hinterlegt sind, die zur Kontrolle des Stream nötig sind.

Wenn die Ausführung eines Programms beginnt, sind die drei Standard-Streams `stdin`, `stdout` und `stderr` bereits automatisch geöffnet.

### Dateioperationen

Die folgenden Funktionen beschäftigen sich mit Datei-Operationen. Der Typ `size_t` ist der vorzeichenlose, ganzzahlige Resultattyp des `sizeof`-Operators.

- `FILE *fopen(const char *filename, const char *mode)`

Öffnet die angegebene Datei; gibt als Ergebnis einen Datenstrom zurück, oder `NULL` falls das Öffnen fehlschlägt.

Als Zugriffsmodus `mode` kann angegeben werden:

- `"r"`: Textdatei zum Lesen öffnen
- `"w"`: Textdatei zum Schreiben neu erzeugen (gegebenenfalls alten Inhalt wegwerfen)

zurück, oder EOF (Konstante mit Wert -1), wenn ein Fehler aufgetreten ist.

- `int fprintf(FILE *stream, const char *format, ...)`

Die Funktion `fprintf()` wandelt Ausgaben um und schreibt sie in `stream` unter Kontrolle von `format`. Als Ergebnis gibt sie die Anzahl der geschriebenen Zeichen zurück; der Wert ist negativ, wenn ein Fehler aufgetreten ist.

- `int printf(const char *format, ...)`

`printf(...)` ist äquivalent zu `fprintf(stdout, ...)`. Die formatierte Ausgabe der `printf()`-Funktion ist im Abschnitt *Ausgabe und Eingabe* näher beschrieben.

- `int sprintf(char *s, const char *format, ...)`

Die Funktion `sprintf()` funktioniert wie `printf()`, nur wird die Ausgabe in das Zeichenarray `s` geschrieben und mit \0 abgeschlossen. `s` muss groß genug für das Resultat sein. Im Ergebniswert wird \0 nicht mitgezählt.

## stdlib.h – Hilfsfunktionen

Die Definitionsdatei `<stdlib.h>` vereinbart Funktionen zur Umwandlung von Zahlen, für Speicherverwaltung und ähnliche Aufgaben.

- `double atof(const char *s)`

Wandelt die Zeichenkette `s` in `double` um. Beendet die Umwandlung beim ersten unbrauchbaren Zeichen.

- `int atoi(const char *s)`

Wandelt die Zeichenkette `s` in `int` um. Beendet die Umwandlung beim ersten unbrauchbaren Zeichen.

- `long atol(const char *s)`

Wandelt die Zeichenkette `s` in `long` um. Beendet die Umwandlung beim ersten unbrauchbaren Zeichen.

- `double strtod(const char *s, char **endp)`

Wandelt den Anfang der Zeichenkette `s` in `double` um, dabei wird Zwischenraum am Anfang ignoriert. Die Umwandlung wird beim ersten unbrauchbaren Zeichen beendet. Die Funktion speichert einen Zeiger auf den eventuell nicht umgewandelten Rest der Zeichenkette bei `*endp`, falls `endp` nicht `NULL` ist. Falls das Ergebnis zu groß ist, (also bei einem Overflow), wird als Resultat `HUGE_VAL` mit dem korrekten Vorzeichen geliefert; liegt das Ergebnis zu dicht bei Null (also bei einem Underflow), wird Null geliefert. In beiden Fällen erhält `errno` den Wert `ERANGE`.

- `long strtol(const char *s, char **endp, int base)`

Wandelt den Anfang der Zeichenkette `s` in `long` um, dabei wird Zwischenraum am Anfang ignoriert. Die Umwandlung wird beim ersten unbrauchbaren Zeichen beendet. Die Funktion speichert einen Zeiger auf den eventuell nicht umgewandelten Rest der Zeichenkette bei `*endp`, falls `endp` nicht `NULL` ist. Hat `base` einen Wert zwischen 2 und 36, erfolgt die Umwandlung unter der Annahme, dass die Eingabe in dieser Basis repräsentiert ist.

Hat `base` den Wert Null, wird als Basis 8, 10 oder 16 verwendet, je nach `s`; eine führende Null bedeutet dabei oktal und `0x` oder `0X` zeigen eine hexadezimale Zahl an. In jedem Fall stehen Buchstaben für die Ziffern von 10 bis `base-1`; bei Basis 16 darf `0x` oder `0X` am Anfang stehen. Wenn das Resultat zu groß werden würde, wird je nach Vorzeichen `LONG_MAX` oder `LONG_MIN` geliefert und `errno` erhält den Wert `ERANGE`.

- `unsigned long strtoul(const char *s, char **endp, int base)`

Funktioniert wie `strtol()`, nur ist der Resultattyp `unsigned long` und der Fehlerwert ist `ULONG_MAX`.

- `int rand(void)`

Gibt als Ergebnis eine ganzzahlige Pseudo-Zufallszahl im Bereich von 0 bis `RAND_MAX` zurück; `RAND_MAX` ist mindestens 32767.

- `void srand(unsigned int seed)`

Benutzt `seed` als Ausgangswert für eine neue Folge von Pseudo-Zufallszahlen. Der erste Ausgangswert ist 1.

- `void * calloc(size_t nobj, size_t size)`

Gibt als Ergebnis einen Zeiger auf einen Speicherbereich für einen Vektor von `nobj` Objekten zurück, jedes mit der Größe `size`, oder `NULL`, wenn die Anforderung nicht erfüllt werden kann. Der Bereich wird mit Null-Bytes initialisiert.

- `void * malloc(size_t size)`

Gibt einen Zeiger auf einen Speicherbereich für ein Objekt der Größe `size` zurück, oder `NULL`, wenn die Anforderung nicht erfüllt werden kann. Der Bereich ist nicht initialisiert.

- `void * realloc(void *p, size_t size)`

Ändert die Größe des Objekts, auf das der Pointer `p` zeigt, in `size` ab. Bis zur kleineren der alten und neuen Größe bleibt der Inhalt unverändert. Wird der Bereich für das Objekt größer, so ist der zusätzliche Bereich nicht initialisiert. `realloc()` liefert einen Zeiger auf den neuen Bereich oder `NULL`, wenn die Anforderung nicht erfüllt werden kann; in diesem Fall wird der Inhalt nicht verändert.

- `void free(void *p)`