Programmieren in C

Veranstaltungsbegleitendes Skript

Gastdozent an der Berliner Hochschule für Technik

Internet: public.bht-berlin.de/~kempfer
E-Mail: gerald.kempfer@bht-berlin.de

Stand: 19. August 2022

Inhaltsverzeichnis

1. Einführung

1.1. Die Geschichte der Sprache

Die Programmiersprache C wurde im wesentlichen Anfang der 70er Jahre von Dennis Ritchie an den Bell Laboratorien entwickelt. Die Entwicklung hing eng mit der Entwicklung von UNIX zusammen. Die Ursprünge von C können zurückverfolgt werden von ALGOL 60 (1960) über Cambridge's CPL (1963) und Martin Richards's BCPL (1967) bis hin zu Ken Thompson's B (1970). Die Sprachen BCPL und deren Nachfolger B waren beide noch sehr assemblernah. UNIX wurde 1970 zunächst noch in B entwickelt, aber die Version 6, die als Betriebssystem 1973 in Betrieb genommen wurde, war weitgehend in C geschrieben. 1978 haben Brian W. Kernighan und Dennis Ritchie das Buch „The C Programming Language" und damit die erste offizielle Beschreibung der Programmiersprache veröffentlicht; daher wurde die Sprache auch häufig K&R C genannt.

Erst 16 Jahre später (1989) wurde vom American National Standards Institute (ANSI) der erste Standard veröffentlicht, der als ANSI-C bzw. auch Standard-C bezeichnet wurde (wird heute auch als C89 bezeichnet). Ein Jahr später übernahm die International Organization For Standardization (ISO) diesen bis dahin rein amerikanischen Standard als internationale Norm und nannte dies C90. Allerdings bezeichnen C89 und C90 den gleichen Standard. 1995 wurden an diesem Standard einige kleine Korrekturen und Erweiterungen vorgenommen; dieser Standard wird „C89 with Amendment 1" (C89 mit Verbesserungen 1) oder kurz C95 genannt. Auch dieser Standard wurde noch einmal gründlich korrigiert und erweitert; dabei entstand 1999 das C99. In diesem Standard sind einige Erweiterungen übernommen worden, die bereits 1993 mit C++ eingeführt wurden. Im weiteren Verlauf der Jahre wurden weitere Korrekturen und Erweiterungen veröffentlicht: Im Dezember 2011 kam C11 mit einer besseren Kompatibilität mit C++ und im Juni 2018 der Standard C18.

Dieses Skript bezieht sich – sofern nicht explizit angegeben – immer auf den C89-Standard.

1.2. Struktur eines C-Programms

Ein Programm besteht aus mehreren Teilen, die in einer oder in verschiedenen Dateien stehen können. Dabei wird im allgemeinen die angegebene Reihenfolge eingehalten.

- Compiler-Instruktionen, dies sind alle Befehle, die mit einem `#` beginnen, sie werden auch Präprozessor-Befehle genannt (z.B. `#include`). Die Präprozessor-Befehle werden in einem späteren Kapitel ausführlich behandelt.
- Deklaration von globalen Objekten
- Funktionsprototypen (Deklaration von Unterprogrammen bzw. von Funktionen)
- Hauptprogramm `int main()`
- Funktionsdefinitionen (Quelltext der Unterprogramme bzw. der Funktionen)

1.3. Das erste einfache Programm

Der erste Schritt in der Programmentwicklung ist immer die Zerlegung des ganzen Problems in kleine Teilprobleme. Diese lassen sich entweder leicht lösen oder in weitere Teilprobleme zerlegen. Jedes Teilproblem wird gelöst und erst einmal in der Umgangssprache formuliert. Als Beispiel nehmen wir mal die Summenberechnung aus zwei Zahlen. Die umgangssprachliche Formulierung lautet dann: *Lies die zwei Zahlen a und b von der Tastatur ein. Dann berechne die Summe beider Zahlen. Zum Schluss gebe das Ergebnis auf dem Bildschirm aus.* Dieses Umsetzen vom Problem in eine umgangssprachliche Lösung ist der schwierigste Teil beim Programmieren. Das anschließende Übersetzen in die Programmiersprache ist dagegen fast wieder einfach.

Unser erstes Programm beinhaltet nur Kommentare; passieren tut noch nichts.

kap01_01.c

```
01 int main()
02 {
03    /* Lies die Zahlen a und b ein              */
04    /* Berechne die Summe von a und b           */
05    /* Gebe das Ergebnis auf dem Bildschirm aus */
06    return 0;
07 }
```

Dabei haben die einzelnen Zeichen/Zeilen folgende Bedeutungen:

`int`	Rückgabewert des Hauptprogramms ist eine ganze Zahl; im allgemeinen wird damit ein Fehlercode an das Betriebssystem zurückgegeben
`main`	Schlüsselwort für das Hauptprogramm
`()`	innerhalb dieser runden Klammern können dem Hauptprogramm Informationen, sogenannte Parameter, mitgegeben werden
`{ }`	Anweisungsblock, innerhalb dieser geschweiften Klammern stehen die Anweisungen an den Rechner
`/* ... */`	Kommentarblock, kann auch über mehrere Zeilen gehen
`return 0;`	Hauptprogramm beenden und Fehlercode 0 (d.h. kein Fehler) zurückgeben

Kommentare werden vom Compiler (das ist ein Programm, das den Programmtext in eine rechnerverständliche Form, d.h. Maschinensprache, übersetzt) vollständig ignoriert. Ein Kommentar, der mit `/* ... */` eingeschlossen ist, kann sich auch über mehrere Zeilen erstrecken. Kommentare sind für den Leser und auch für den Autor sehr wichtig. Sie erläutern die Parameter, die Voraussetzungen sowie die Funktionsweise des Programms. Wird in einem Team programmiert, sind auch Änderungsdatum und Name des Autors wichtig.

- Der Programmtext selber ist eine reine Textdatei, die mit jedem Texteditor bearbeitet werden kann.
- Es gibt Schlüsselworte (z.B. `main`), die grundsätzlich klein geschrieben werden (mit einigen Ausnahmen). Generell wird zwischen Groß- und Kleinschreibung unterschieden!
- Die Zeilenstruktur ist für den Compiler unwichtig. Nur für die Lesbarkeit sollte eine gewisse Struktur eingehalten werden. Auch Leerzeilen sind für die Lesbarkeit sehr wichtig. Mit ihnen lassen sich logische Abschnitte voneinander trennen.

Im nächsten Schritt werden die Anweisungen für das obige Problem eingefügt, für das wir bereits eine umgangssprachliche Lösung entwickelt hatten. Dabei wird **jede** Anweisung (außer den Präprozessor-Befehlen) mit einem Semikolon beendet.

kap01_02.c

```
01 #include <stdio.h>
02
03 int main()
04 {
05    int a, b, summe;
06
07    /* Lies die Zahlen a und b ein              */
08    printf("Geben Sie bitte zwei ganze Zahlen ein:");
09    scanf("%i %i", &a, &b);
10
11    /* Berechne die Summe von a und b           */
12    summe = a + b;
13
```

```
14      /* Gebe das Ergebnis auf dem Bildschirm aus */
15      printf("Summe: %i", summe);
16
17      return 0;
18  }
```

Folgendes ist jetzt neu hinzugekommen:

`#include <stdio.h>`	Einbinden der Ein-/Ausgabefunktionen. Diese Zeile muss in jedem Programm stehen, das Eingaben von der Tastatur erwartet (Funktion `scanf`) oder Ausgaben auf den Bildschirm bringt (Funktion `printf`). Die Datei `stdio.h` wird auch **Headerdatei** genannt.
`int a,b,summe;`	*Deklaration* und *Definition* von Variablen vom Typ "`int`" ("integer", ganze Zahlen). Der Compiler stellt daraufhin für die Variablen entsprechenden Speicherplatz zur Verfügung. Ab hier kennt der Compiler die drei Variablen. Mit Hilfe der Variablennamen kann auf die Werte der Variablen zugegriffen werden.
`printf`	`printf` ist die Standardausgabe auf dem Bildschirm. In Klammern wird der Text in Anführungsstrichen angegeben, der auf dem Bildschirm ausgegeben werden soll.
`scanf`	`scanf` ist die Standardeingabe von der Tastatur. In Klammern wird zuerst das Eingabeformat in Anführungsstrichen angegeben ("`%i %i`", d.h. es sollen zwei ganze Zahlen eingelesen werden), dahinter ein Verweis auf die Variable, die den eingegebenen Wert aufnehmen soll (`&a`). Für diesen Verweis wird vor dem Variablennamen ein kaufmännisches Und (`&`) gesetzt. Sollen mit einem `scanf`-Befehl Werte für mehrere Variablen eingelesen werden, so müssen die Verweise auf die Variablen mit Kommata getrennt werden.
`=`	Zuweisung: Der Variablen auf der linken Seite (**l-value** bzw. **L-Wert**) des Gleichheitszeichens wird das Ergebnis des Ausdrucks auf der rechten Seite zugewiesen.
"*Text*"	beliebige Zeichenkette (im englischen: string), die die Anführungszeichen selbst nicht enthalten darf. Wenn die Zeichenkette die Anführungszeichen enthalten soll, sind sie als `\"` zu schreiben.
`;`	beendet jede Deklaration und jede Anweisung

Damit haben wir bereits das erste Programm geschrieben.

2. Zeichen, Zeichensätze und Tokens

2.1. Zeichen

Ein C-Programm ist eine Folge von Zeichen aus einem Zeichensatz. Üblicherweise werden dabei folgende Zeichen verwendet:

1. Die 52 Groß- und Kleinbuchstaben:
   A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
   a b c d e f g h i j k l m n o p q r s t u v w x y z
2. Die 10 Ziffern:
   0 1 2 3 4 5 6 7 8 9
3. Das Leerzeichen
4. Der horizontaler und der vertikaler Tabulator sowie der Seitenvorschub
5. Die folgenden 29 Zeichen:
   ! " % & ( ) = . , : ; + - * / # ~ { } [ ] < > | ' _ \ ? ^

Zusätzlich wird ein Zeichen oder eine Zeichenfolge benötigt, die das Ende der Programmzeilen angibt. Alle weiteren Zeichen (z.B. für die Formatierung des Quelltextes) werden als Leerzeichen angesehen und beeinflussen den Quelltext in keiner Weise.

2.2. Zeichensatz im ausführenden Programm

Der Zeichensatz, der für das Programm während der Ausführung verwendet wird, muss nicht identisch sein mit dem Zeichensatz, der für die Eingabe der Quelltexte verwendet wird. Daher kann es bei der Ein- und Ausgabe zu unerwünschten Effekten kommen. So werden z.B. bei einem Konsolen-Programm, das unter MS Windows eingegeben und in einem DOS-Fenster ausgeführt wird, die deutschen Umlaute in der Bildschirmausgabe nicht korrekt angezeigt.

Um diese Probleme zu umgehen, müssen diese Zeichen als ASCII-Werte (oktal) in den Quelltext eingegeben werden.

Beispiel:

kap02_01.c

```
01 #include <stdio.h>
02
03 int main()
04 {
05    printf("Deutsche Umlaute\n\n");
06
07    printf("Linux-Konsole (ISO-8859):\n");
08    printf("ae = \344\n");
09    printf("oe = \366\n");
10    printf("ue = \374\n\n");
11
12    printf("DOS-Eingabeaufforderung:\n");
13    printf("ae = \204\n");
14    printf("oe = \224\n");
15    printf("ue = \201\n\n");
16
17    return 0;
18 }
```

2.3. Weiße Leerzeichen und Zeilenende

In einem C-Programm werden das Leerzeichen, das Zeilenende, der horizontale und der vertikaler Tabulator und der Seitenvorschub als **Weiße Leerzeichen** (**White Spaces**) bezeichnet. Auch Kommentare werden wie weiße Leerzeichen behandelt (siehe Abschnitt *Kommentare*). Diese Zeichen werden ignoriert, da sie nur zum Trennen von benachbarten Tokens dienen – mit Ausnahme, wenn diese Zeichen in konstanten Zeichen bzw. Zeichenketten stehen.

Weiße Leerzeichen werden im allgemeinen verwendet, um ein C-Programm für uns Menschen lesbarer zu machen (siehe Kapitel *Gestaltung von C-Programmen*).

Das Zeilenende-Zeichen bzw. -Zeichenfolge markiert das Ende einer Quelltextzeile. Manche C-Compiler interpretieren die Eingabetaste, den Seitenvorschub und den vertikalen Tabulator auch als Ende einer Quelltextzeile. Dies ist wichtig für das Erkennen, wann eine Präprozessorzeile (siehe Kapitel *Präprozessorbefehle*) zu Ende ist.

Eine Quelltextzeile kann in der nächsten Editor-Zeile fortgeführt werden, wenn sie mit einem Backslash ('\') oder mit dem Trigraph '??/' (siehe nächsten Abschnitt *Trigraphen*) beendet wird. Im C99-Standard lassen sich damit sogar Tokens aufsplitten. Beim Compilieren wird der Backslash bzw. der Trigraph samt Zeilenende beseitigt, um eine längere, logisch zusammenhängende Zeile zu erzeugen. Dies geschieht nach dem Konvertieren der Trigraphen und der Multibyte-Zeichen, aber noch vor dem Präprozessor und dem eigentlichen Compiliervorgang.

Beispiel:

Die Zeilen

```
if (a == b) x = 1; el\
se x = 2;
```

werden umgewandelt in

```
if (a == b) x = 1; else x = 2;
```

Viele C-Compiler lassen nur eine bestimmte Länge für die Quelltextzeilen – vor und nach dem Umwandeln – zu. Der C89-Standard erlaubt logische Zeilen bis maximal 509 Zeichen, der C99-Standard bis maximal 4.095 Zeichen.

2.4. Trigraphen

Mit dem Standard-C wurden einige Zeichenfolgen eingeführt, mit denen wichtige Zeichen dargestellt werden können, auch wenn der Quelltext mit einem Zeichensatz geschrieben wird, der diese Zeichen nicht beinhaltet. Da diese Zeichenfolgen immer aus drei Zeichen bestehen, werden diese auch **Trigraphen** genannt. Trigraphen werden auch in konstanten Zeichenfolgen erkannt und umgesetzt. Das Umwandeln der Trigraphen erfolgt beim Aufruf des Compilers als erstes, sogar noch vor dem Präprozessor. Es gibt genau neun Trigraphen, alle anderen Zeichenfolgen, die mit zwei Fragezeichen beginnen, werden nicht verändert.

Trigraph	Ersetzt	Trigraph	Ersetzt
??(	[	??)	]
??<	{	??>	}
??/	\	??!	|
??'	^	??-	~
??=	#		

Beispiel:

Um eine konstante Zeichenfolge mit einem Backslash auszugeben, müssen zwei aufeinander folgende Backslashs geschrieben werden. Jeder dieser Backslash kann wieder in einen Trigraphen umgesetzt werden. Daher wird mit der Zeichenfolge "??/??/" ein Backslash "\" ausgegeben.

Um zu verhindern, dass eine Zeichenfolge als Trigraph interpretiert wird, muss das zweite Fragezeichen durch '\?' ersetzt werden. Um die Zeichenkette "Wer??!" zu erhalten, muss also "Wer?\?!" eingegeben werden.

kap02_02.c

```
01 #include <stdio.h>
02
03 int main()
04 {
05    printf("Trigraphen:\n\n");
06
07    printf("Backslash: ??/??/\n");
08    printf("Wer??!   oder   Wer?\?!\n");
09
10    return 0;
11 }
```

Bei manchen Compiler muss explizit angegeben werden, dass Trigraphen verwendet werden. Zum Beispiel muss beim GNU-C-Compiler `gcc` zusätzlich die Option `-trigraphs` angegeben werden.

2.5. Multibytes und Wide Characters

Mit C95 wurden die Multibytes und Wide Characters eingeführt. Damit können auch Zeichensätze verwendet werden, die mehr als 256 Zeichen beinhalten. Auf diese speziellen Zeichen wird im Kapitel *Datentypen in C* weiter eingegangen.

2.6. Kommentare

Kommentare werden vom Compiler komplett ignoriert, d.h. sie werden wie weiße Leerzeichen behandelt. Kommentare sollen zum Verständnis des Quelltextes dienen bzw. dem Leser wichtige Hinweise geben.

Ein Kommentar beginnt mit den zwei Zeichen `/*` und endet mit den zwei Zeichen `*/`. Dazwischen können beliebig viele Zeichen – einschließlich Zeilenumbrüche – stehen. Im Standard-C können Kommentare nicht geschachtelt werden.

In C99 wurden zusätzlich Zeilenkommentare eingeführt. D.h. ein Kommentar kann auch mit `//` beginnen. Dieser Kommentar endet dann automatisch am Ende der Zeile.

Kommentare werden nicht innerhalb von konstanten Zeichen oder Zeichenketten – also innerhalb von Anführungszeichen – erkannt.

Kommentare werden noch vor dem Präprozessor entfernt, so dass Präprozessorbefehle innerhalb von Kommentaren nicht erkannt werden können. Auch haben Zeilenumbrüche innerhalb von Kommentaren keinerlei Auswirkungen auf die Programmzeile, in der der Kommentar steht.

Standard-C schreibt vor, dass alle Kommentare durch ein einzelnes Leerzeichen ersetzt werden sollen. Einige ältere Compiler löschen wohl die Kommentare, aber setzen dafür kein Leerzeichen ein. Dies kann fatale Folgen haben, da ohne Leerzeichen die Texte vor und hinter dem Kommentar nahtlos zusammengeschrieben werden.

Kommentare sollten nicht dafür genutzt werden, um größere Programmteile auszukommentieren. Denn ist in diesem Programmteil selber wieder ein Kommentar (geschachtelte Kommentare; siehe oben), wird nicht wirklich das ganze Programmteil auskommentiert. Ein Compilerfehler ist die Folge.

2.7. Tokens

Die Zeichen, aus denen das C-Programm besteht, werden in sogenannten Tokens unterteilt. Es gibt fünf Gruppen von Tokens: Operatoren, Trennzeichen, Bezeichner, Schlüsselwörter und Literale.

Benachbarte Tokens werden am besten durch weiße Leerzeichen getrennt. Vor und nach Operatoren müssen keine weißen Leerzeichen stehen; der Compiler setzt dann von links nach rechts immer die längstmöglichen Tokens zusammen, auch wenn das Ergebnis kein gültiges C-Programm mehr ist. Von daher sollten auch hier weiße Leerzeichen verwendet werden.

Beispiele:

In der folgenden Tabelle werden verschiedene Zeichenfolgen und deren Erkennung in C-Tokens dargestellt. Dabei werden die einzelnen erkannten C-Tokens mit einem Komma getrennt.

Zeichenfolge	C-Tokens
`forwhile`	`forwhile`
`b>x`	`b, >, x`
`b->x`	`b, ->, x`
`b--x`	`b, --, x`
`b---x`	`b, --, -, x`

Im ersten Beispiel werden die Schlüsselwörter `for` und `while` nicht als einzelne Tokens erkannt, da sie nicht mit einem weißen Leerzeichen getrennt wurden.

Das vierte Beispiel erzeugt einen Fehler beim Compilieren, da hier die Tokens falsch erkannt werden. Richtig erkannt werden sie erst mit dem Einsetzen von weißen Leerzeichen: `b - -x`. Dann wird nämlich das erste Minuszeichen als Operator für die Subtraktion und das zweite Minuszeichen als Vorzeichen für die Variable `x` erkannt.

Das fünfte Beispiel ist für den Leser nicht klar: Es kann `b-- - x` oder `b - --x` gemeint sein. Erst durch die Regel, dass von links nach rechts immer die längstmöglichen Tokens erkannt werden, macht klar, dass der Compiler tatsächlich die erste Variante erkennt. Wird die zweite Variante gewünscht, müssen entsprechende weiße Leerzeichen eingefügt werden.

2.8. Operatoren und Trennzeichen

Die im Standard-C gültigen Operatoren und Trennzeichen sind in der folgenden Tabelle aufgelistet. Um Programmierer zu unterstützen, deren Tastatur die geschweiften und eckigen Klammern und die Raute (`#`) nicht enthalten, können auch die alternativen Schreibweisen verwendet werden: `<%`, `%>`, `<:`, `:>`, `%:` und `%:%:`.

Im traditionellen C werden die zusammengesetzten Zuweisungsoperatoren als zwei seperate Tokens erkannt – nämlich als Operator und als Gleichheitszeichen. Hier dürfen also auch weiße Leerzeichen zwischen Operator und Gleichheitszeichen verwendet werden. Im Standard-C dagegen werden die zusammengesetzten Zuweisungsoperatoren als ein einzelnes Token erkannt; entsprechend dürfen im Standard-C keine weißen Leerzeichen zwischen Operator und Gleichheitszeichen eingefügt werden.

Bezeichnung	Tokens	
Einzelne Operatoren	`! % ^ & * - + = ~ | . < > / ?`	
Zusammengesetzte Zuweisungsoperatoren	`+= -= *= /= %= <<= >>= &= ^= |=`	
Andere zusammengesetzte Operatoren	`-> ++ -- << >> <= >= == != && ||`	
Trennzeichen	`( ) [ ] { } , ; : ...`	
Alternative Schreibweise für Trennzeichen	`<%`	(geschweifte Klammer auf)
	`%>`	(geschweifte Klammer zu)
	`<:`	(eckige Klammer auf)
	`:>`	(eckige Klammer zu)
	`%:`	(Raute)
	`%:%:`	(zwei Rauten hintereinander)

Beispiel:

kap02_03.c

```
01 %:include <stdio.h>
02
03 int main()
04 <%
05    int Array<:10:> = <% 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 %>;
06
07    printf("Array[5] = %i\n", Array<:5:>);
08
09    return 0;
10 %>
```

Die alternativen Schreibweisen für Trennzeichen werden nicht von allen Compilern unterstützt.

2.9. Bezeichner

Bezeichner sind Namen für Konstanten, Variablen, Datentypen, Klassen, Funktionen oder Makros. Sie bestehen aus einer Folge von Buchstaben, Ziffern und dem Unterstrich; sie dürfen allerdings nicht mit einer Ziffer beginnen und dürfen keine Schlüsselwörter sein. Dabei wird bei den Buchstaben zwischen Groß- und Kleinschreibung unterschieden, d.h. die Bezeichner `abc` und `ABC` sind zwei unterschiedliche Bezeichner.

Seit C99 dürfen Bezeichner auch andere universelle oder systemabhängige Multibyte-Zeichen beinhalten. Diese dürfen wie die Ziffern nicht als erstes Zeichen verwendet werden und müssen ähnlich wie Buchstaben sein. Eine Liste der für Bezeichner erlaubten Zeichen ist im C99-Standard ISO/IEC 9899: 1999 und im ISO/IEC TR 10176-1998 zu finden. Darauf wird in diesem Skript nicht weiter eingegangen.

Neben den Schlüsselwörtern muss man aufpassen, nicht einen Bezeichner der Standard-Bibliotheken doppelt zu verwenden. Viele dieser Bezeichner fangen mit einem Unterstrich an, gefolgt von entweder einem zweiten Unterstrich oder einem Großbuchstaben. Daher sollten keine Bezeichner verwendet werden, die mit einem Unterstrich beginnen.

Während Bezeichner im Prinzip beliebig lang sein dürfen – sie dürfen maximal so lang sein, wie eine Programmzeile lang sein darf –, wird nur eine bestimmte (signifikante) Anzahl von Zeichen zur Unterscheidung der Bezeichner verwendet. Vor dem C89-Standard betrug sie manchmal nur 8 Zeichen (Compilerabhängig; schließlich gab es damals noch keinen Standard), im C89-Standard betrug sie mindestens 31 Zeichen und im C99-Standard mindestens 63 Zeichen.

Beispiel:

Vor dem C89-Standard waren für einen Compiler die beiden Bezeichner `LangerBezeichner` und `LangerBezeichnerNeu` identisch, da nur die ersten 8 Zeichen zur Unterscheidung verwendet wurden.

Für externe Bezeichner – also Bezeichner, die mit dem Schlüsselwort `extern` deklariert werden – gelten weitere Einschränkungen, da diese Bezeichner unter Umständen auch von anderen Compilern, Linkern oder Debugger, die stärkere Begrenzungen haben, bearbeitet werden müssen. So fordert der C89-Standard vom Compiler, dass externe Bezeichner eine maximale Länge von mindestens 6 Zeichen haben dürfen. Der C99-Standard hat dies auf mindestens 31 Zeichen erweitert.

2.10. Schlüsselwörter

Die Bezeichner, die in der folgenden Tabelle aufgelistet werden, sind **Schlüsselwörter** (engl. *keywords*) und dürfen anderweitig nicht als Bezeichner (z.B. als Variablen) verwendet werden. Schlüsselwörter dürfen aber als Präprozessor-Makronamen verwendet werden, da der Präprozessor diese Makronamen umsetzt bevor der Compiler mit der Erkennung der Schlüsselwörter beginnt.

`auto`	`_Bool` [1]	`break`	`case`	`char`	`_Complex` [1]
`const`	`continue`	`default`	`do`	`double`	`else`

```
enum          extern      float      for        goto       if
_Imaginary [1] inline [1]  int        long       register   restrict [1]
return        short       signed     sizeof     static     struct
switch        typedef     union      unsigned   void       volatile
while
```

[1] Diese Schlüsselwörter sind erst im C99-Standard enthalten. Davon ist aber nur das Schlüsselwort `inline` auch in C++ ein Schlüsselwort!

Mit dem C11-Standard sind noch folgende Schlüsselwörter dazu gekommen:

```
_Alignas    _Alignof        _Atomic         _Generic
_Noreturn   _Static_assert  _Thread_local
```

Zusätzlich haben einige Compiler als Spracherweiterung die Schlüsselwörter `asm` (Schlüsselwort in C++) und `fortran`. Ferner sind in der Headerdatei `iso646.h` die Makros `and`, `and_eq`, `bitand`, `bitor`, `compl`, `not`, `not_eq`, `or`, `or_eq`, `xor` und `xor_eq` definiert, die in C++ zu den Schlüsselwörtern gehören.

2.11. Literale

Es gibt vier verschiedene Arten von **Literalen** (engl. *literals*): Ganze Zahlen, Fließkommazahlen, Zeichen und Zeichenketten. Manchmal werden Literale auch fälschlicherweise "Konstante" genannt; sie müssen aber von den unveränderlichen Variablen, Konstanten und Datentypen unterschieden werden.

Jedes Literal ist gekennzeichnet durch Wert und Datentyp. Die Formate der verschiedenen Literale werden im Kapitel *Datentypen in C* vorgestellt.

3. Variablen

In unserem ersten Programm im Kapitel *Einführung* sind `a`, `b` und `summe` veränderliche Daten und heißen **Variablen**. Sie sind Objekte eines vordefinierten Grunddatentyps für ganze Zahlen (`int`, Kurzform für integer). Der Begriff "Variable" wird für ein veränderliches Objekt gebraucht.

Dabei bestehen Variablen im Wesentlichen aus zwei Teilen: Aus einem **Wert** und aus einer **Adresse**, an der der Wert gespeichert wird. Zusätzlich haben Variablen einen **Datentyp** und können (müssen aber nicht!) einen **Namen** haben.

3.1. Deklaration von Variablen

Variablen müssen deklariert werden. Die Zeile

```
int summe, a, b;
```

ist eine **Deklaration**. Unter Deklaration wird verstanden, dass der Variablenname dem Compiler bekannt gemacht wird. Wenn dieser Name später im Programm versehentlich falsch geschrieben wird, kennt der Compiler den Namen nicht und gibt eine Fehlermeldung aus. Damit dienen Deklarationen unter anderem der Programmsicherheit.

3.2. Definition von Variablen

Damit Variablen benutzt werden können, müssen sie auch definiert werden. Die gleiche Zeile wie oben

```
int summe, a, b;
```

ist auch gleichzeitig eine **Definition** der drei Variablen. Unter Definition wird verstanden, dass für die Variablen ein Speicherbereich reserviert wird. Variablen belegen damit Bereiche im Arbeitsspeicher des Computers, deren Inhalte während des Programmlaufs verändert werden können. Die Variablennamen sind also symbolische Adressen, unter denen der Wert gefunden wird und über die auf den Inhalt, also auf den Wert zugegriffen werden kann.

Deklaration und Definition werden unterschieden, weil es auch Deklarationen ohne gleichzeitige Definition geben kann, doch davon später mehr (im Kapitel *Funktionen*). Zunächst sind die Deklarationen zugleich Definitionen.

Wenn beispielsweise für `a` eine 100 und für `b` eine 2 eingegeben wird, sieht der Speicher nach dem Programmlauf wie folgt aus. Dabei sind die Adressen willkürlich gewählt.

Adresse	Variablenname	Inhalt
10120		0
10124		17
10128	a	100
10132	b	2
10136	summe	102
10140		4009

Wichtig: Vor C99 mussten die Variablen immer am Anfang einer Funktion deklariert und definiert werden.

3.3. Initialisierung von Variablen

Variablen haben nach der Definition einen beliebigen Wert (je nachdem, was vorher in dieser Speicherzelle stand). Sie können vor der Benutzung **initialisiert** werden, d.h. sie erhalten einen definierten Anfangswert. Dies geschieht grundsätzlich bei der Definition der Variablen.

Beispiel:
Definition der Variablen `a`, `b`, `c` mit gleichzeitiger Initialisierung der Variablen `a` und `b` auf die Werte 2 bzw. 3. Auch die Variable `c` wird initialisiert mit dem Ergebnis der Rechenoperation `a + b`.

```
int a = 2, b = 3, c = a + b;
```

Von der Ausführung her ist dies auf den ersten Blick dasselbe wie die folgenden Zeilen:

```
int a, b, c;
a = 2;
b = 3;
c = a + b;
```

Im letzten Fall werden die Variablen `a`, `b` und `c` aber nicht initialisiert, sondern die Werte werden den Variablen zugewiesen! Eine Zuweisung ist also keine Initialisierung und umgekehrt, obwohl in beiden Fällen das Gleichheitszeichen als Operator verwendet wird.

Eine Initialisierung kann nur bei der gleichzeitigen Definition (also der Erzeugung) einer Variablen auftreten, eine Zuweisung setzt immer ein schon vorhandenes Objekt voraus!

Werden dagegen die Variablen `a` und `b` nicht initialisiert, ist das Ergebnis von `c` nicht vorhersehbar. Im allgemeinen geben die Compiler dabei eine Warnung aus, dass nicht initialisierte Variablen im Programm verwendet werden.

3.4. Unveränderliche Variablen und Konstanten

Eine **unveränderliche Variable** wird mit dem Schlüsselwort `const` (wird daher auch häufig fälschlicherweise als Konstante bezeichnet) definiert. Dieses Schlüsselwort gibt es erst seit dem C89-Standard. Unveränderliche Variablen müssen bei der Definition sofort initialisiert werden. Denn zu einem späteren Zeitpunkt darf die unveränderliche Variable nicht mehr verändert werden – weder per Zuweisung noch durch Inkrementator oder Dekrementator. Der Compiler prüft dies, um sicherzustellen, dass der Programmierer nicht ausversehens eine unveränderliche Variable ändert. Einige Compiler - wie z.B. gcc - geben allerdings nur eine Warnung aus.

Das Schlüsselwort `const` kann dabei wahlweise vor oder nach dem Datentypen geschrieben werden. Während die Schreibweise mit `const` vor dem Datentypen häufig zu sehen ist, wäre die andere Schreibweise deutlich besser für die Lesbarkeit des Programms.

Beispiel:

```
const int a = 37;
int const b = 25;

a = 5;    /* Fehler! */
b++;      /* Fehler! */
```

Konstanten werden z.B. durch den Präprozessor-Befehl `#define` (siehe Kapitel *Präprozessor-Befehle*) definiert. Konstante haben nur einen Namen und einen Wert; anders als (unveränderliche) Variablen, die zusätzlich noch eine Adresse (einen Speicherort) haben, und anders als Literale, die nur einen Wert haben.

Desweiteren sind die Namen von Arrays und Funktionen auch Konstanten, denn ihre Werte geben an, an welcher Adresse das Array bzw. die Funktion steht, aber sie haben selber keine Adresse.

3.5. Ausdrücke und Werte

Ein **Ausdruck** ist eine *syntaktische* Größe – z.B. eine Variable oder ein Literal.

Ein **Wert** ist eine *semantische* Größe, die während der Programmausführung ermittelt oder berechnet wird.

Werte werden immer durch Ausdrücke beschrieben oder anders herum gesagt: Ein Ausdruck bezeichnet einen Wert (der unter Umständen erst errechnet werden muss).

3.6. L-Werte und R-Werte

Jede Variable besitzt einen **L-Wert** und einen **R-Wert**. Dabei ist die Adresse der Variablen der L-Wert und der Wert der Variable der R-Wert. Die Buchstaben L und R stehen für Links und Rechts und sind relativ zum Zuweisungsoperator zu verstehen. Ein L-Wert steht demnach immer links und ein R-Wert immer rechts vom Zuweisungsoperator.

Beispiel:

```
int Zahl1 = 1, Zahl2 = 2;

Zahl1 = Zahl2;
```

Bei dieser Zuweisung wird der Wert der Variablen `Zahl2` (also der R-Wert) an der Adresse der Variablen `Zahl1` (also der L-Wert) gespeichert.

Ausdrücke, die L-Werte beschreiben, werden **L-Ausdrücke** und Ausdrücke, die R-Werte beschreiben, werden **R-Ausdrücke** genannt. Zum Beispiel ist der Name einer Variablen ein L-Ausdruck, während ein Literal ein R-Ausdruck ist.

Ein L-Ausdruck ist dabei mehr als ein R-Ausdruck, denn ein L-Ausdruck beinhaltet L- und R-Werte, während R-Ausdrücke nur R-Werte liefern können. Ein L-Ausdruck liefert automatisch immer dann einen L-Wert, wenn er links vom Zuweisungsoperator steht, und immer dann einen R-Wert, wenn er rechts vom Zuweisungsoperator steht. Im obigen Beispiel sind beide Variablen `Zahl1` und `Zahl2` L-Ausdrücke. Die Variable `Zahl1` steht links vom Gleichheitszeichen, daher wird hier der L-Wert verwendet; die Variable `Zahl2` steht aber rechts vom Gleichheitszeichen, daher wird hier der R-Wert verwendet.

R-Ausdrücke sind vor allem Literale, Ausdrücke, die sich mit Operatoren zusammensetzen (außer der verkürzten `if`-Abfrage `?:` (siehe Kapitel *Kontrollstrukturen*) und dem Variablen-Operator (siehe Kapitel *Zeiger*), sowie Funktionsaufrufe.

Achtung:

Der Name einer jeden Variablen ist wohl ein L-Ausdruck, aber nicht jeder Name von Variablen darf links vom Zuweisungsoperator stehen: Unveränderlichen Variablen (z.B. `const int u;`) dürfen keine Werte zugewiesen werden. Trotzdem sind die Namen von unveränderlichen Variablen L-Ausdrücke, da auch unveränderliche Variablen eine Adresse (L-Wert) und einen Wert (R-Wert) haben.

3.7. Gültigkeitsbereich

Der **Gültigkeitsbereich** (engl. *scope*) ist der Bereich im C-Quelltext, in dem eine deklarierte Variable sichtbar bzw. bekannt ist. Es wird zwischen folgenden Gültigkeitsbereichen unterschieden (weitere Gültigkeitsbereiche werden im Laufe des Skriptes vorgestellt):

Globale Variablen werden außerhalb von Funktionen – also auch außerhalb der `main`-Funktion – deklariert (**Top Level**). Sie sind vom Deklarationspunkt (siehe Abschnitt *Deklarationspunkt*) bis zum Ende des C-Quelltextes bekannt.

Lokale Variablen (manchmal auch Block-Variablen genannt) werden innerhalb von Funktionen bzw. innerhalb eines Blockes deklariert. Sie sind vom Deklarationspunkt bis zum Ende der Funktion bzw. des Blockes bekannt.

Beispiel:

kap03_01.c

```
01 int Global;         /* globale Variable */
02
03 int main()
04 {
05    int Lokal;       /* lokale Variable
06                        innerhalb von main*/
07    {
08       int Block;    /* lokale Variable
09                        innerhalb des Blocks */
10       Block = 9;
11    } /* Ende des Gültigkeitsbereiches
12         für die Variable Block! */
13
14    Lokal = 5;
15    Global = 7;
16    Block = 3;       /* Fehler! */
17
18    return 0;
19 } /* Ende des Gültigkeitsbereiches
20      für die Variable Lokal! */
21
22 /* Ende des Gültigkeitsbereiches
23    für die Variable Global! */
```

Damit dieses Programm ohne Fehler compiliert werden kann, muss die Zeile 16 auskommentiert werden!

3.8. Deklarationspunkt

Eine Variable kann im Normalfall immer erst dann verwendet werden, wenn sie komplett deklariert ist. Um festzulegen, wann eine Variable komplett deklariert ist, wird ein sogenannter **Deklarationspunkt** definiert: Eine Variable ist komplett deklariert nach dem letzten Token des Variablennamens. Danach – also auch innerhalb der gleichen Zeile – kann die Variable verwendet werden.

Beispiel:

```
int intsize = sizeof(intsize);
/*          ^
   Deklarationspunkt
*/
```

In diesem Beispiel kann die Variable `intsize` mit ihrer eigenen Größe initialisiert werden, da die Initialisierung hinter dem Deklarationspunkt liegt.

3.9. Sichtbarkeit

Eine Deklaration einer Variable ist überall innerhalb seines Gültigkeitsbereiches sichtbar. Die Sichtbarkeit kann aber durch eine Deklaration einer weiteren, gleichnamigen Variable überlappen, so dass die erste Variable nicht mehr sichtbar ist; sie ist sozusagen versteckt. Voraussetzung ist, dass die zweite Variable innerhalb eines Blocks deklariert wird, der im Gültigkeitsbereich der ersten Variable liegt.

Beispiel:

```
int Var;         /* globale Variable */

int main()
```

```
{
   float Var;    /* damit wird die obige Variable "versteckt" */
   ...
}
```

Eine Variable ist immer erst ab der Stelle gültig, an der sie deklariert wird – und nicht ab Beginn des Blockes, in dem sie deklariert ist. Dadurch kann es – wie im folgenden Beispiel – zu Situationen kommen, in denen innerhalb eines Blockes gleichnamige Variablen auf unterschiedliche Deklarationen zurückzuführen sind. Nach Standard-C ist dies durchaus korrekt, aber die Verwendung gleichnamiger Variablen innerhalb eines Blockes ist ein schlechter Programmierstil!

Beispiel:

kap03_02.c

```
01 #include <stdio.h>
02
03 int Var;         /* globale Variable */
04
05 int main()
06 {
07    Var = 1;      /* gemeint ist die
08                     globale Variable */
09    printf("globale Variable Var = %i\n", Var);
10
11    float Var;    /* lokale Variable  */
12
13    Var = 1.5;    /* jetzt ist die
14                     lokale Variable gemeint */
15    printf("lokale Variable Var = %f\n", Var);
16
17    return 0;
18 }
```

3.10. Speicherklassen

Die Angabe einer Speicherklasse bestimmt den Gültigkeitsbereich mit und gibt ferner an, ob die deklarierte Variable auch für den Linker bekannt sein soll. Es gibt die folgenden fünf Speicherklassen:

`auto` Kann nur bei der Deklaration von Variablen innerhalb von Funktionen und Blöcken verwendet werden. Die deklarierte Variable ist dadurch eine lokale (automatische) Variable und ist nur innerhalb des Blockes gültig, in dem sie deklariert wurde (vom Deklarationspunkt bis zum Ende des Blockes). Da diese Speicherklasse der Standard ist, kann das Schlüsselwort `auto` auch weggelassen werden – das ist auch der Grund, warum dieses Schlüsselwort nur selten in C-Programmen zu finden ist.

Seit dem C++11-Standard hat dieses Schlüsselwort eine neue Bedeutung erhalten; daher wird dringend empfohlen, diese Speicherklasse in C nicht mehr zu verwenden.

`extern` Kann für globale oder lokale Variablen verwendet werden. Durch dieses Schlüsselwort wird angegeben, dass die deklarierte Variable in einer anderen C-Quelltextdatei deklariert und definiert ist. Erst beim Linken wird die deklarierte Variable mit der definierten Variable (aus einer anderen C-Quelltextdatei) verbunden. Standardmäßig werden externe Variablen als globale Variablen deklariert und sind daher innerhalb der ganzen Quelltextdatei (vom Deklarationspunkt bis zum Ende der Quelltextdatei) gültig. Als lokale Variablen sind externe Variablen auch nur innerhalb des Blockes gültig, in dem sie deklariert wurden. Einige C-Compiler, die sich nicht ganz an das Standard-C halten, sehen alle externen Variablen als globale Variablen.

`register` Kann nur bei der Deklaration von lokalen Variablen oder bei Funktions-Parametern verwendet werden. Der Compiler bekommt damit den Hinweis, dass diese Variable häufig genutzt wird und dass die Variable – wenn möglich – so definiert werden sollte, dass die Zugriffszeit möglichst gering ist. Der schnellste Zugriff wird erreicht, wenn die Variable in einem Register des Prozessors definiert wird.

`static` Kann für globale oder lokale Variablen verwendet werden. Statische Variablen sind wohl nur innerhalb des Blockes (lokal) bzw. innerhalb der Quelltextdatei (global) sichtbar, in denen sie deklariert wurden; sie sind dann aber bis zum Ende des Programms gültig! D.h. am Ende des Gültigkeitsbereiches von normalen Variablen werden statische Variablen nicht vernichtet und behalten sogar ihren Wert bei. Statische Variablen mit Initialisierung werden dadurch auch nur beim einmaligen Anlegen initialisiert; wird der Block mit der statischen Variablen-Deklaration erneut aufgerufen, existiert diese Variable bereits und die Initialsierung wird nicht durchgeführt.

Beispiel:

kap03_03.c

```
01 #include <stdio.h>
02
03 int main()
04 {
05    int i;
06
07    for (i = 0; i < 5; i++)
08    {
09       static int Statisch = 0;
10       int NichtStatisch = 0;
11
12       Statisch = Statisch + 1;
13       NichtStatisch = NichtStatisch + 1;
14       printf("%i: Statisch = %d; ", i, Statisch);
15       printf("NichtStatisch = %d\n", NichtStatisch);
16    }
17
18    return 0;
19 }
```

Der Anweisungsblock der Schleife wird fünf Mal durchlaufen; dabei werden folgende Werte ausgegeben:

```
0: Statisch = 1; NichtStatisch = 1
1: Statisch = 2; NichtStatisch = 1
2: Statisch = 3; NichtStatisch = 1
3: Statisch = 4; NichtStatisch = 1
4: Statisch = 5; NichtStatisch = 1
```

Statische Variablen werden ferner nicht dem Linker bekannt gegeben, d.h. es können keine externen Verweise auf statische Variablen deklariert werden.

`typedef` Anders als bei den anderen vier Speicherklassen wird mit diesem Schlüsselwort ein neuer Datentyp aus vorhandenen Datentypen definiert. Anstelle eines Variablennamens wird der Name des neuen Datentyps angegeben.

Beispiel:

```
typedef int GanzeZahl; /* GanzeZahl ist jetzt ein Datentyp  */
GanzeZahl i;           /* Variable i vom Datentyp GanzeZahl */
```

3.11. volatile

Mit dem Schlüsselwort `volatile` (engl. für *flüchtig*) bei der Variablendefinition wird dem C-Compiler mitgeteilt, dass der Wert dieser Variable auch von außerhalb des Programms – also von anderen

Programmen bzw. von der Hardware – geändert werden kann. Solche Variablen werden bei der Optimierung des Quellcodes vom Compiler nicht berücksichtigt.

Am häufigsten wird dieser Typspezifizierer beim Zugriff auf Speicheradressen der Computer-Hardware verwendet, da die Hardware den Wert der Speicheradresse unabhängig vom Programm ändern kann (z.B. Ein- und Ausgabepuffer sowie Kontrollregister).

4. Datentypen in C

4.1. Ausdruck

Ein Ausdruck besteht aus einem oder mehreren Operanden, die miteinander durch Operatoren verknüpft sind. Der Ausdruck hat einen Wert als Ergebnis, der an die Stelle des Ausdrucks tritt. Der einfachste Ausdruck besteht aus einer einzigen Konstanten oder Variablen. Die Operatoren müssen zu den Datentypen der Operanden passen.

Beispiele:
```
25
1 + 2
"Text"
```

Der Ausdruck `a = 1 + 2` ist ein zusammengesetzter Ausdruck. Die Operanden `1` und `2` sind Zahlenwerte (Literale), die durch den +-Operator verknüpft werden. Der resultierende Wert `3` tritt nun an die Stelle des Ausdrucks `1 + 2` und wird der Variablen `a` zugewiesen. Das Ergebnis des gesamten Ausdrucks ist der Wert von `a`, also `3`.

Daraus folgt, dass in einem Ausdruck mehrere Zuweisungen an Variablen erfolgen können, z.B. `a = b = c = 1 + 2`. Der Wert `3` wird zuerst der Variablen `c` zugewiesen. Das Ergebnis der Zuweisung ist der neue Wert von `c`, also `3`. Dieser Wert wird nun der Variablen `b` zugewiesen. Das Ergebnis dieser Zuweisung wird schließlich der Variablen `a` zugewiesen. Das Ergebnis dieser Zuweisung wird nicht weiter verarbeitet, da links vom Ausdruck `a` nichts mehr steht.

4.2. Ganze Zahlen

Es gibt mehrere verschiedene Repräsentationen von ganzen Zahlen, die sich durch die Anzahl der verwendeten Bits unterscheiden. Diese heißen `short`, `int`, `long` und `long long`. Letzterer Datentyp wurde erst mit C99 eingeführt. Hierbei gilt: Anzahl Bits von `short` <= Anzahl Bits von `int` <= Anzahl Bits von `long` <= Anzahl Bits von `long long`. Standard-C schreibt nur die Mindestanzahl der verwendeten Bits vor. Typische Werte für ein System mit einem 32-Bit-Prozessor sind in der folgenden Tabelle dargestellt. Die tatsächlichen Werte für Ihr System finden Sie in der Headerdatei `limits.h`.

Datentyp	Mindestanzahl Bits	Typische Anzahl von Bits
`short`	16 Bits	16 Bits
`int`	16 Bits	32 Bits
`long`	32 Bits	32 Bits
`long long`	64 Bits	64 Bits

In der folgenden Tabelle werden die alternativen Schreibweisen für die verschiedenen Datentypen der ganzen Zahlen aufgeführt:

Datentyp	alternative Schreibweisen		
`short`	`short int`	`signed short`	`signed short int`
`int`	`signed int`	`signed`	
`long`	`long int`	`signed long`	`signed long int`
`long long`	`long long int`	`signed long long`	`signed long long int`

Das Schlüsselwort `signed` wurde erst mit C89 eingeführt und sollte aus Kompatibilitätsgründen mit älteren Compilern weggelassen werden.

Entsprechend der Anzahl der verwendeten Bits lassen sich unterschiedliche Zahlenbereiche darstellen:

16 Bits:	-2^{15}	...	$+2^{15} - 1$	entspricht	-32.768	... +32.767
32 Bits:	-2^{31}	...	$+2^{31} - 1$	entspricht	-2.147.483.648	... +2.147.483.647
64 Bits:	-2^{63}	...	$+2^{63} - 1$	entspricht	-9.223.372.036.854.775.808	... +9.223.372.036.854.775.807

Wie oben bereits erwähnt, sind die tatsächlichen Zahlenbereiche für jedes System in der Headerdatei `limits.h` als Konstante hinterlegt. Das folgende Beispielprogramm zeigt, wie Sie sich diese Konstanten anzeigen lassen können.

kap04_01.c

```
01 #include <stdio.h>
02 #include <limits.h>
03
04 int main()
05 {
06    printf("CHAR_BIT   = %i\n"  , CHAR_BIT  );   // Anzahl Bits fuer ein Byte
07    printf("SCHAR_MIN  = %i\n"  , SCHAR_MIN );   // min. Wert (signed char)
08    printf("SCHAR_MAX  = %i\n"  , SCHAR_MAX );   // max. Wert (signed char)
09    printf("UCHAR_MAX  = %i\n"  , UCHAR_MAX );   // max. Wert (unsigned char)
10    printf("CHAR_MIN   = %i\n"  , CHAR_MIN  );   // min. Wert (char)
11    printf("CHAR_MAX   = %i\n"  , CHAR_MAX  );   // max. Wert (char)
12    printf("MB_LEN_MAX = %i\n"  , MB_LEN_MAX);   // max. Anzahl Bytes
13                                                 // fuer ein Multibytezeichen
14    printf("SHRT_MIN   = %i\n"  , SHRT_MIN  );   // min. Wert (short)
15    printf("SHRT_MAX   = %i\n"  , SHRT_MAX  );   // max. Wert (short)
16    printf("USHRT_MAX  = %i\n"  , USHRT_MAX );   // max. Wert (unsigned short)
17    printf("INT_MIN    = %i\n"  , INT_MIN   );   // min. Wert (int)
18    printf("INT_MAX    = %i\n"  , INT_MAX   );   // max. Wert (int)
19    printf("UINT_MAX   = %u\n"  , UINT_MAX  );   // max. Wert (unsigned int)
20    printf("LONG_MIN   = %li\n" , LONG_MIN  );   // min. Wert (long)
21    printf("LONG_MAX   = %li\n" , LONG_MAX  );   // max. Wert (long)
22    printf("ULONG_MAX  = %lu\n" , ULONG_MAX );   // max. Wert (unsigned long)
23    printf("LLONG_MIN  = %lli\n", LLONG_MIN );   // min. Wert (long long)
24    printf("LLONG_MAX  = %lli\n", LLONG_MAX );   // max. Wert (long long)
25    printf("ULLONG_MAX = %llu\n", ULLONG_MAX);   // max. Wert (unsigned long long)
26
27    return 0;
28 }
```

Die Zahlen werden im Rechner als Binärzahlen dargestellt. Dabei werden positive Zahlen mit führender 0 und negative Zahlen mit führender 1 dargestellt. Um eine positive Zahl zu negieren, werden alle Bits der Zahl invertiert und anschließend eine 1 addiert. Dieses Verfahren wird **Zweier-Komplement-Darstellung** (engl: *two-complement notation*) genannt. Um beispielsweise die Zahl 2 (8Bit binär: 00000010_2) zu negieren, werden erst alle Bits invertiert (8Bit binär: 11111101_2) und anschließend eine 1 addiert. Das Ergebnis lautet dann -2 (8Bit binär: 11111110_2).

Zwei Zahlen ändern sich nicht, wenn sie negiert werden: 0 und die kleinste negative Zahl. Wenn bei der 0 (8Bit binär: 00000000_2) alle Bits negiert werden (8Bit binär: 11111111_2) und anschließend eine 1 addiert wird, ergibt dies 100000000_2. Da es nun aber 9 Bits sind, wird für die Darstellung in 8 Bit das linke Bit "abgeschnitten" und übrig bleibt die 00000000_2. Wird die kleinste negative Zahl (8 Bit binär: 10000000_2 = -128 dezimal) negiert, ergibt sich nach der Negation aller Bits als Zwischenergebnis eine 01111111_2 und nach der Addition einer 1 die Ausgangszahl 10000000_2.

Es gibt noch zwei weitere Verfahren, um Zahlen zu negieren, die beide vom Standard-C akzeptiert werden: Die **Einer-Komplement-Darstellung** (engl. *ones-complement notation*) und die **Vorzeichen-Wert-Darstellung** (engl. *sign magnitude notation*).

In der Einer-Komplement-Darstellung werden – wie bei der Zweier-Komplement-Darstellung – beim Negieren alle Bits negiert, aber es wird keine 1 mehr addiert. Dadurch reduziert sich der Zahlenbereich um eins: Mit beispielsweise 16 Bits können dann nur noch die Zahlen von -32767 bis +32767 dargestellt werden, d.h. die kleinste negative Zahl -32768 entfällt. Dafür gibt es zwei verschiedene Zahlen, die beide den Wert 0 haben: +0 (binär: 00000000 00000000_2) und -0 (binär: 11111111 11111111_2).

In der Vorzeichen-Wert-Darstellung wird beim Negieren nur das Vorzeichen-Bit negiert; die restlichen Bits bleiben unverändert. Genau wie bei der Einer-Komplement-Darstellung ist der Zahlenbereich um eins

reduziert, da auch hier die kleinste negative Zahl -32768 wegfällt. Für die Zahl 0 gibt es auch hier zwei verschiedene Darstellungsmöglichkeiten: +0 (binär: 00000000 00000000_2) und -0 (binär: 10000000 00000000_2).

Im weiteren Verlauf werden auf diese beiden alternativen Darstellungsmöglichkeiten für negative Zahlen nicht weiter eingegangen und statt dessen von der üblichen Zweier-Komplement-Darstellung ausgegangen.

Durch das Schlüsselwort `unsigned` wird gekennzeichnet, dass es sich um rein positive Zahlen handelt. Das Bit, das sonst für das Vorzeichen benötigt wird, wird hierbei nun für einen größeren Zahlenbereich verwendet. Die zulässigen Bereiche für ganze positive Zahlen sehen dann wie folgt aus:

16 Bits: 0 ... $+2^{16} - 1$ entspricht 0 ... +65.535
32 Bits: 0 ... $+2^{32} - 1$ entspricht 0 ... +4.294.967.295
64 Bits: 0 ... $+2^{64} - 1$ entspricht 0 ... +18.446.744.073.709.551.615

Beim Rechnen mit ganzen Zahlen ist die *begrenzte Genauigkeit* zu beachten, denn die oben angegebenen Zahlenbereiche sind ja nur eine Untermenge der ganzen Zahlen. Daraus folgt, dass das Ergebnis einer Folge von arithmetischen Operationen nur dann korrekt ist, wenn kein Zwischenergebnis den durch den Datentyp vorgegebenen maximalen Zahlenbereich überschreitet.

kap04_02.c

```
01 #include <stdio.h>
02
03 int main()
04 {
05    short a = 50;       /* a,b,c sind short-Zahlen,         */
06    short b = 1000;     /* also 16 Bit-Zahlen               */
07    short c;            /* => -32.768 ... 32.767            */
08
09    c = a * b;
10    printf("%i * %i = ", a, b);
11    printf("%i\n", c); /* Ausgabe: -15.536 statt 50.000 ! */
12
13    return 0;
14 }
```

Grundsätzlich immer den größten Datentyp zu verwenden (nach dem Motto: Sicher ist sicher!), ist nicht sinnvoll, weil Variablen dann mehr Speicherplatz und auch mehr Rechenzeit benötigen.

Ganze Zahlen können in drei Zahlensystemen verwendet werden:

- **Oktalzahlen**: Wenn eine Zahl mit einer 0 beginnt, wird sie als Oktalzahl interpretiert, z.B. 0377 = $377_8 = 3 * 8^2 + 7 * 8^1 + 7 * 8^0 = 255_{10}$ (dezimal).
- **Hexadezimalzahlen**: Wenn eine Zahl mit "0x" oder "0X" beginnt, wird sie als Hexadezimalzahl interpretiert, z.B. 0XAFFE = 45054 dezimal.
- **Dezimalzahlen**

4.3. Operatoren für ganze Zahlen

Auf Daten eines bestimmten Typs kann man nur bestimmte Operationen durchführen. Eine Zeichenkette kann beispielsweise nicht mit einer anderen multipliziert werden. Also**: Ein Operand und der zugehörige Operator gehören zusammen!**

Im folgenden werden alle Operatoren für ganze Zahlen aufgelistet:

Arithmetische Operatoren:

Operator	Beispiel	Bedeutung
+	+i	unäres Plus (kann weggelassen werden)
-	-i	unäres Minus

++	++i	vorherige Inkrementierung (Erhöhung um 1)
	i++	nachfolgende Inkrementierung
--	--i	vorherige Dekrementierung (Erniedrigung um 1)
	i--	nachfolgende Dekrementierung
+	i + 2	binäres Plus (Addition)
-	i - 5	binäres Minus (Subtraktion)
*	5 * i	Multiplikation
/	i / 6	Division
%	i % 4	Modulo (Divisionsrest)
=	i = 3 + j	Zuweisung

Arithmetische Kurzform-Operatoren:

Operator	Beispiel	Bedeutung
+=	i += 3	i = i + 3
-=	i -= 3	i = i - 3
*=	i *= 3	i = i * 3
/=	i /= 3	i = i / 3
%=	i %= 3	i = i % 3

relationale Operatoren:

Operator	Beispiel	Bedeutung
<	i < 3	kleiner als
>	i > 3	größer als
<=	i <= 3	kleiner als oder gleich
>=	i >= 3	größer als oder gleich
==	i == 3	gleich
!=	i != 3	ungleich

4.4. Bitoperatoren

Weil ganze Zahlen auch als *Bitvektoren* aufgefasst werden können, sind zusätzlich Bitoperationen möglich. Dabei werden die Zahlen in Binärdarstellung betrachtet.

Beispiele:
```
int a = 5, b;
b = a << 2;
```

Dies bewirkt eine Bitverschiebung um 2 Stellen nach links, wobei von rechts Nullen nachgezogen werden (beim Verschieben nach rechts werden von links Nullen nachgezogen; bei vorzeichenbehafteten Zahlen wird - je nach Compiler - entweder eine Null oder das Vorzeichen nachgezogen). Dies entspricht der Multiplikation mit 2^2, also mit 4.

`0000 0000 0000 0101` binäre Darstellung der Zahl 5
`0000 0000 0001 0100` alle Bits um 2 Stellen nach links verschoben

Die Variable `a` hat nun den Wert 20 (0000 0000 0001 0100_2).

```
a = a & b;
```

Diese Anweisung bewirkt eine bitweise UND-Verknüpfung, d.h. das Ergebnis-Bit ist 1, wenn die Bits der beiden Operanden auch gleich 1 sind, ansonsten 0. In der binären Darstellung sind das wie folgt aus:

`0000 0000 0001 0100` binäre Darstellung der Zahl 20
`0000 0000 0000 0101` binäre Darstellung der Zahl 5
`0000 0000 0000 0100` bitweises UND; Ergebnis: 4

In den folgenden zwei Tabellen werden alle Bitoperatoren aufgelistet.

Bitoperatoren:

Operator	Beispiel	Bedeutung
`<<`	`i << 2`	Bit nach links schieben (Multiplikation mit 2er-Potenzen)
`>>`	`i >> 1`	Bit nach rechts schieben (Division durch 2er-Potenzen)
`&`	`i & 7`	bitweises UND
`^`	`i ^ 7`	bitweises Exklusiv-ODER (XOR)
`|`	`i | 7`	bitweises ODER
`~`	`~i`	bitweises Negieren

Bit-Kurzform-Operatoren:

Operator	Beispiel	Bedeutung
`<<=`	`i <<= 2`	`i = i << 2`
`>>=`	`i >>= 1`	`i = i >> 1`
`&=`	`i &= 3`	`i = i & 3`
`^=`	`i ^= 3`	`i = i ^ 3`
`|=`	`i |= 3`	`i = i | 3`

4.5. Reele Zahlen

Reelle Zahlen, auch Fließkomma- oder Gleitkommazahlen genannt, bauen sich folgendermaßen auf:

- Vorzeichen (optional)
- Vorkommastellen
- Dezimalpunkt (**KEIN** Komma!)
- Nachkommastellen
- `e` oder `E` und Ganzzahl-Exponent (optional)
- Suffix `f`,`F` oder `l`,`L` (optional)
  (`f`,`F` für `float`; `l`,`L` für `long double`; Zahlen ohne Suffix sind vom Typ `double`)

Einige Beispiele für reelle Zahlen sind:

```
-236.265e6f
3.4E3
3.1415
1e-08
9.2L
```

Mit dem Exponenten sind Zehnerpotenzen gemeint, d.h. `3.4E3` ist also identisch mit `3.4 * 10`^{3} oder `3400`.

Reelle Zahlen werden durch drei Datentypen dargestellt. Sie sind ähnlich wie bei den ganzen Zahlen durch Zahlenbereiche eingeschränkt, bedingt durch die Anzahl der verwendeten Bits pro Zahl. Die angegebenen Bits sowie die Genauigkeiten sind beispielhaft, da sie von System zu System variieren.

Typ	Bit	Zahlenbereich	Stellen Genauigkeit
`float`	32	+/-1.17549 * 10^{-38} ... +/-3.40282 * 10^{38}	7
`double`	64	+/-2.22507 * 10^{-308} ... +/-1.79769 * 10^{308}	15
`long double`	80	+/-3.3621 * 10^{-4932} ... +/-1.18973*10^{4932}	19

Genauso wie bei den ganzen Zahlen gibt es auch für die Fließkommazahlen eine Headerdatei, in der die Zahlenbereiche für das aktuelle System als Konstante hinterlegt sind. Diese Headerdatei heißt `float.h`. Im folgenden Beispielprogramm wird gezeigt, wie Sie sich diese Konstanten anzeigen lassen können.

kap04_03.c

```
01 #include <stdio.h>
02 #include <float.h>
03
04 int main()
05 {
06    // FLT: float;   DBL: double;   LDBL: long double
07    printf("FLT_RADIX       = %i\n" , FLT_RADIX      );   // Basis fuer Exponentendarst.
08    printf("FLT_MANT_DIG    = %i\n" , FLT_MANT_DIG   );   // Anzahl Stellen Mantisse
09    printf("DBL_MANT_DIG    = %i\n" , DBL_MANT_DIG   );
10    printf("LDBL_MANT_DIG   = %i\n" , LDBL_MANT_DIG  );
11    printf("FLT_DIG         = %i\n" , FLT_DIG        );   // Genauigkeit Dezimalziffern
12    printf("DBL_DIG         = %i\n" , DBL_DIG        );
13    printf("LDBL_DIG        = %i\n" , LDBL_DIG       );
14    printf("FLT_MIN_EXP     = %i\n" , FLT_MIN_EXP    );   // min. neg. FLT_RADIX-Exp.
15    printf("DBL_MIN_EXP     = %i\n" , DBL_MIN_EXP    );
16    printf("LDBL_MIN_EXP    = %i\n" , LDBL_MIN_EXP   );
17    printf("FLT_MIN_10_EXP  = %i\n" , FLT_MIN_10_EXP );   // min. neg. 10er-Exponent
18    printf("DBL_MIN_10_EXP  = %i\n" , DBL_MIN_10_EXP );
19    printf("LDBL_MIN_10_EXP = %i\n" , LDBL_MIN_10_EXP);
20    printf("FLT_MAX_EXP     = %i\n" , FLT_MAX_EXP    );   // max. FLT_RADIX-Exponent
21    printf("DBL_MAX_EXP     = %i\n" , DBL_MAX_EXP    );
22    printf("LDBL_MAX_EXP    = %i\n" , LDBL_MAX_EXP   );
23    printf("FLT_MAX_10_EXP  = %i\n" , FLT_MAX_10_EXP );   // max. 10er-Exponent
24    printf("DBL_MAX_10_EXP  = %i\n" , DBL_MAX_10_EXP );
25    printf("LDBL_MAX_10_EXP = %i\n" , LDBL_MAX_10_EXP);
26    printf("FLT_MAX         = %g\n" , FLT_MAX        );   // max. Fließkommawert
27    printf("DBL_MAX         = %g\n" , DBL_MAX        );
28    printf("LDBL_MAX        = %Lg\n", LDBL_MAX       );
29    printf("FLT_EPSILON     = %g\n" , FLT_EPSILON    );   // kleinster Wert, fuer den
30    printf("DBL_EPSILON     = %g\n" , DBL_EPSILON    );   // 1.0 + x ungleich 1.0 gilt
31    printf("LDBL_EPSILON    = %Lg\n", LDBL_EPSILON   );
32    printf("FLT_MIN         = %g\n" , FLT_MIN        );   // min. normal. Fließkommawert
33    printf("DBL_MIN         = %g\n" , DBL_MIN        );
34    printf("LDBL_MIN        = %Lg\n", LDBL_MIN       );
35
36    return 0;
37 }
```

Eine beliebige Genauigkeit ist allerdings nicht für alle Zahlen möglich! Für die Darstellung der reellen Zahlen mit 32 Bit beispielsweise existieren nur 2^{32} = 4.294.967.296 verschiedene Möglichkeiten, eine Zahl zu bilden. Ein reelles Zahlenkontinuum ist das nun nicht gerade und alle Illusionen von der computertypischen Genauigkeit und Korrektheit sind über den Haufen geschmissen! Mögliche Folgen der nicht exakten Darstellung können sein:

- Werden zwei fast gleich große Werte subtrahiert, heben sich die signifikanten Ziffern auf und das Ergebnis ist ungenau (**numerische Auslöschung**).
- Die Division durch betragsmäßig zu kleine Werte hat einen **Überlauf** (**Overflow**) zum Ergebnis, d.h. das Ergebnis liegt außerhalb des Zahlenbereiches des Datentyps. Ähnlich ist es bei der **Unterschreitung** (**Underflow**). Diese tritt auf, wenn das Ergebnis zu klein ist, als das es mit dem gegebenen Datentyp dargestellt werden kann. Das Ergebnis wird dann auf 0 gesetzt.
- Die Reihenfolge einer Berechnung kann entscheidend sein! Wenn beispielsweise die drei Variablen a, b und c addiert werden sollen, ist es mathematisch gesehen kein Unterschied, ob zuerst a und b addiert und

zu diesem Ergebnis c addiert wird oder ob zuerst b und c addiert und dann a dazuaddiert wird. Anders dagegen beim Computer und der Programmiersprache C/C++. Andere Programmiersprachen haben die gleichen oder ähnliche Probleme. Daher muss zu kritischen Berechnungen auch immer eine Genauigkeitsbetrachtung gemacht werden!

Beispiel für Probleme mit der Rechengenauigkeit:

kap04_04.c

```
01 #include <stdio.h>
02
03 int main()
04 {
05    float a = 1.234567E-9f, b = 1.000000f, c = -b;
06    float s1, s2;
07
08    s1 = a + b;
09    s1 += c;
10    s2 = b + c;
11    s2 += a;
12    printf("%e\n", s1); /* Erg.: 0.000000e+00 */
13    printf("%e\n", s2); /* Erg.: 1.234567e-09 */
14
15    return 0;
16 }
```

Der interne Aufbau einer Fließkommazahl ist durch den IEEE Standard for Binary Floating-Point Arithmetic (ISO/IEEE Standard 754-1985) oder kurz IEEE Standard 754 festgelegt. Dieser wird im folgenden anhand des Datentyps `float` vorgestellt:

Bei einer 32Bit-`float`-Zahl ist das ganz linke Bit für das Vorzeichen *S*, die nächsten 8 Bits von links (Bit 2 bis 9) bilden den vorzeichenlosen Exponenten *E* und die restlichen 23 Bits die Mantisse *M* (nicht vergleichbar mit einer Mantisse im mathematischen Sinne). Die Zahl *f* wird dann wie folgt berechnet:

$f = (-1)^S \cdot 2^{(E-127)} \cdot (1 + \sum_{i=1}^{23} M_i \cdot 2^{-i})$, wobei M_i das i-te Bit der Mantisse von links ist.

Beispiel:

`float`-Zahl binär dargestellt: $00111101\ 11001100\ 11001100\ 11001101_2$

Dann ist *S* gleich 0, *E* gleich 01111011_2 (gleich 123 dezimal) und *M* gleich $10011001100110011001101_2$. Diese Werte werden in die Formel eingesetzt:

$$f = (-1)^0 \cdot 2^{(123-127)} \cdot (1 + \frac{1}{2^1} + \frac{1}{2^4} + \frac{1}{2^5} + \frac{1}{2^8} + \frac{1}{2^9} + \frac{1}{2^{12}} + \frac{1}{2^{13}} + \frac{1}{2^{16}} + \frac{1}{2^{17}} + \frac{1}{2^{20}} + \frac{1}{2^{21}} + \frac{1}{2^{23}})$$

$$f = 2^{-4} \cdot (1 + \frac{1}{2} + \frac{1}{16} + \frac{1}{32} + \frac{1}{256} + \frac{1}{512} + \frac{1}{4096} + \frac{1}{8192} + \frac{1}{65536} + \frac{1}{131072} + \frac{1}{1048576} + \frac{1}{2097152} + \frac{1}{8388608})$$

$$f = \frac{1}{16} \cdot (1{,}600015342469308890548406862745 1)$$

$$f = 0{,}100000958904331805659275428921 56$$

Bei sieben Stellen Genauigkeit (1 Stelle vor und 6 Stellen nach dem Komma) ergibt diese Zahl also 0,1.

Es sind nun noch einige Sonderfälle zu betrachten:

Exponent *E* gleich 255:

Mantisse *M* ungleich 0:	*f* = NaN (Not a Number)
Mantisse *M* gleich 0:	*f* = +/- Infinity (Unendlich; entsprechend des Vorzeichens *S*)

Exponent *E* gleich 0:

Mantisse *M* ungleich 0: Zahl ist nicht normalisiert und lässt sich wie folgt berechnen:

$$f = (-1)^S \cdot 2^{(E-126)} \cdot \sum_{i=1}^{23} M_i \cdot 2^{-i}$$

Mantisse *M* gleich 0: f = +/- 0 (entsprechend des Vorzeichens *S*)

4.6. Operatoren für reelle Zahlen

Die Operatoren für reelle Zahlen sind die folgenden:

Arithmetische Operatoren:

Operator	Beispiel	Bedeutung
+	+f	unäres Plus (kann weggelassen werden)
-	-f	unäres Minus
+	f + 2	binäres Plus (Addition)
-	f - 5	binäres Minus (Subtraktion)
*	5 * f	Multiplikation
/	f / 6	Division
=	f = 3 + g	Zuweisung

Arithmetische Kurzform-Operatoren:

Operator	Beispiel	Bedeutung
+=	f += 3	f = f + 3
-=	f -= 3	f = f - 3
*=	f *= 3	f = f * 3
/=	f /= 3	f = f / 3

relationale Operatoren:

Operator	Beispiel	Bedeutung
<	f < 3	kleiner als
>	f > 3	größer als
<=	f <= 3	kleiner als oder gleich
>=	f >= 3	größer als oder gleich
==	f == 3	gleich
!=	f != 3	ungleich

4.7. Regeln zum Bilden von Ausdrücken

Es gelten im allgemeinen die Regeln der Algebra beim Berechnen eines Ausdrucks, z.B. Klammerregeln und Punkt- vor Strichrechnung. In der folgenden Tabelle werden die Prioritäten der einzelnen Operatoren aufgelistet. Dabei ist die Priorität 1 die höchste und 16 die niedrigste Priorität.

Priorität	Operator	Reihenfolge
1	[] () . -> ++ -- (postfix) {}	links nach rechts

2	++ -- (prefix) sizeof ~ ! - + (unär, d.h. als Vorzeichen) & (Adressoperator) * (Variablenoperator)	rechts nach links
3	(type name) (Typkonvertierung)	rechts nach links
4	* / %	links nach rechts
5	+ -	links nach rechts
6	<< >>	links nach rechts
7	< > <= >=	links nach rechts
8	== !=	links nach rechts
9	& (bitweises UND)	links nach rechts
10	^ (bitweises Exklusiv-ODER)	links nach rechts
11	| (bitweises ODER)	links nach rechts
12	&& (logisches UND)	links nach rechts
13	|| (logisches ODER)	links nach rechts
14	?:	rechts nach links
15	alle Zuweisungen wie =, +=, -=, ...	rechts nach links
16	,	links nach rechts

Auf gleicher Prioritätsstufe wird ein Ausdruck von links nach rechts abgearbeitet. Dies wird auch *linksassoziativ* genannt. Ausnahme: Die unären und Zuweisungsoperatoren werden von rechts nach links abgearbeitet (*rechtsassoziativ*). Generell werden aber zuerst immer die Klammern ausgewertet.

Beispiele:
`a = b + c + d;` ist gleich mit `a = ((b + c) + d);`
`a = b = c = d;` ist gleich mit `a = (b = (c = d));`

Die Auswertungsreihenfolge der Teilausdrücke einer Priorität untereinander ist jedoch nicht festgelegt und kann von jedem Compiler anders gesetzt werden. Daher sollte es vermieden werden, in einem Ausdruck einen Wert gleichzeitig zu verändern und zu benutzen, wie es die folgenden Beispiele zeigen:

Beispiele:
```
int Teil = 0;
Summe = (Teil = 3) + (++Teil);
```

Das Ergebnis von Summe kann sowohl den Wert `4` als auch den Wert `7` annehmen, je nachdem welche Klammer zuerst ausgewertet wird.

```
int i = 2;
i = 3 * i++;
```

Erste Möglichkeit: Es wird `3 * i` berechnet und das Ergebnis `6` wird der Variablen `i` zugewiesen. Anschließend wird `i` um 1 erhöht. Das Endergebnis ist dann `7`. Zweite Möglichkeit: Es wird `3 * i` berechnet. Nun wird `i` um 1 (von `2` auf `3`) erhöht. Anschließend wird aber `i` das Ergebnis der Berechnung `3 * i` zugewiesen. Das Endergebnis ist in diesem Fall `6`.

4.8. Zeichen

Zeichen sind Buchstaben wie `a`, `B`, `C`, `d`, Ziffernzeichen wie `4`, `5`, `6` und Sonderzeichen wie `;,.!` sowie andere Zeichen. Für sie gibt es den Datentyp `char`. Der Datentyp Zeichen beinhaltet immer nur ein Zeichen, das durch eine 1-Byte-Zahl (ganze Zahl) intern gespeichert wird. Daraus folgt, dass es 256 verschiedene

Zeichen geben kann. Davon sind die ersten 128 international festgelegt, während die anderen 128 regional unterschiedlich sind (diese werden für nationale Sonderzeichen genutzt). Der Zusammenhang zwischen den Zeichen und den intern gespeicherten Zahlen ist in der sogenannten ASCII-Tabelle festgelegt (siehe Kapitel 14 im Skript „*Grundlagen der Informatik*“).

Konstante Zeichen werden in Hochkommata eingeschlossen, also beispielsweise `'y'`, `'9'`, `'?'`.

Hinweis: Speziell bei den Ziffernzeichen muss zwischen dem Zeichen (z.B. `'1'`) und der Ziffer (z.B. `1`) unterschieden werden!

Es wird generell zwischen den Datentypen `signed char` (interner Zahlenbereich: -128 ... +127) und `unsigned char` (interner Zahlenbereich: 0 ... 255) unterschieden. Meist wird aber nur `char` verwendet, wobei damit bei den meisten Compilern der Datentyp `signed char` gemeint ist.

Da Zeichen intern als ganze Zahlen gespeichert werden, können alle Operatoren der ganzen Zahlen auch auf Zeichen angewendet werden, wobei nicht alle Operationen auch Sinn machen (beispielsweise die Addition zweier Zeichen). Daher noch einmal eine Tabelle mit den Operatoren, die für Zeichen sinnvoll sind (`a` ist eine Variable vom Typ `char`).

Operator	Beispiel	Bedeutung
=	`a = 'X'`	Zuweisung
<	`a < 'z'`	kleiner als
>	`a > 'c'`	größer als
<=	`a <= 'M'`	kleiner als oder gleich
>=	`a >= 'k'`	größer als oder gleich
==	`a == 'J'`	gleich
!=	`a != 'n'`	ungleich

Es gibt besondere Zeichenkonstanten, die nicht direkt gedruckt bzw. auf dem Bildschirm angezeigt werden können. Um diese darzustellen, werden sie als zwei Zeichen geschrieben, benötigen aber wie alle Zeichen nur ein Byte. Das erste der zwei Zeichen ist ein Backslash (`'\'`). Diese Zeichen werden auch **Escape-Sequenzen** genannt, weil der Backslash als Escape-Zeichen verwendet wird, um der normalen Interpretation als ein einzelnes Zeichen zu entkommen (to escape). In der nächsten Tabelle werden einige dieser Escape-Sequenzen aufgelistet.

Zeichen	Bedeutung
`\a`	Signalton
`\f`	Seitenvorschub
`\n`	neue Zeile
`\r`	Zeilen- oder Wagenrücklauf
`\t`	Tabulator
`\\`	Backslash \
`\'`	Hochkomma '
`\"`	Anführungszeichen "
`\0`	Nullbyte (z.B. für String-Ende)
`\?`	Fragezeichen ? (C99)

4.9. Multibyte und Wide Characters

Ein Wide Character wird durch den Datentyp `wchar_t` dargestellt und ist auf den meisten Systemen als `int` definiert (in der Headerdatei `stddef.h`). Ein Array von Wide Characters ist ein Wide String.

4.10. Logischer Datentyp

Vor dem C99-Standard gab es keinen direkten logischen Datentypen. Statt dessen werden die logischen Werte durch ganze Zahlen dargestellt. Dabei wird eine ganze Zahl ungleich Null als logischen Wert wahr (im englischen: *true*) und eine ganze Zahl gleich Null als falsch (im englischen: *false*) interpretiert. Ergebnisse von logischen Operationen sind gleich 1 für wahr und gleich 0 für falsch.

Beispiel:
```
int i = 17, j;
j = !i;   /* ergibt 0 (falsch), da i ungleich 0 (wahr) ist */
i = !j;   /* ergibt 1 (wahr), da j gleich 0 (falsch) ist   */
```

Erst mit C99 wurde ein logischer Datentyp eingeführt: `_Bool`. Aber auch dieser Datentyp ist eine ganze Zahl, der allerdings nur die Zahlen 0 und 1 (für falsch und wahr) speichern kann.

Zusätzlich wurde mit dem C99-Standard die Headerdatei `stdbool.h` eingeführt. In ihr werden das Makro `bool` als Datentyp sowie die Konstanten `false` und `true` (0 und 1) definiert. Dies sind Definitionen und keine Schlüsselwörter (im Gegensatz zum Datentyp `_Bool`)! Dies ist gerade für ältere C-Programme wichtig, da in diesen meistens der Datentyp `bool` mit den Konstanten `false` und `true` vom Programmierer selber definiert wurde. In diesen Fällen sollte diese Headerdatei nicht eingefügt werden.

Da logische Variablen intern als ganze Zahlen gespeichert werden, können alle Operatoren der ganzen Zahlen hier auch angewendet werden, wobei nicht alle Operationen auch Sinn machen (beispielsweise die Addition wahr und wahr). Daher noch einmal eine Tabelle mit den Operatoren, die für logische Variablen sinnvoll sind.

Operator	Beispiel	Bedeutung
`!`	`!i`	logische Negation
`&&`	`i && j`	logisches UND
`||`	`i || j`	logisches ODER
`=`	`h = i && j`	Zuweisung

Werden mehrere Bedingungen mit einem logischen Und oder einem logischen Oder verknüpft, ist zu beachten, dass die Compiler im allgemeinen versuchen, die Ermittlung des Ergebnisses der verknüpften Bedingungen zu optimieren. Das bedeutet, dass bei einem logischen Und die zweite Bedingung nur dann ausgewertet wird, wenn die erste Bedingung wahr ist. Denn ist die erste Bedingung falsch, ist beim logischen Und auch das Ergebnis der Verknüpfung falsch, ganz gleich wie die zweite Bedingung aussieht. Und bei einem logischen Oder wird die zweite Bedingung nur dann ausgewertet, wenn die erste Bedingung falsch ist. Denn ist die erste Bedingung wahr, ist beim logischen Oder auch das Ergebnis der Verknüpfung unabhängig von der zweiten Bedingung wahr. Dabei können unangenehme Seiteneffekte entstehen!

Im folgenden Beispiel wird diese Optimierung ausgenutzt:

kap04_05.c

```
01 #include <stdio.h>
02
03 int main()
04 {
05    int a = 0, b = 1;
06    char Zeichen = 'A';
07
08    (a && (Zeichen = 'B'));
09    printf("Zeichen = %c\n", Zeichen);
10
11    (b || (Zeichen = 'C'));
12    printf("Zeichen = %c\n", Zeichen);
13
14    return 0;
15 }
```

Durch die Optimierung werden die beiden Zuweisungen der Zeichen `'B'` und `'C'` nicht ausgeführt und es wird

```
Zeichen = A
Zeichen = A
```

ausgegeben. Diese Optimierung ist aber nicht vorgeschrieben. Und ohne Optimierung würden die beiden Zuweisungen ausgeführt werden und das Ergebnis sieht anders aus. Von daher sollte das Programm wie folgt umgeschrieben werden, um in jedem Fall das obige Ergebnis zu erhalten.

kap04_06.c

```
01 #include <stdio.h>
02
03 int main()
04 {
05    int a = 0, b = 1;
06    char Zeichen = 'A';
07
08    if (a)
09       Zeichen = 'B';
10    printf("Zeichen = %c\n", Zeichen);
11
12    if (!b)
13       Zeichen = 'C';
14    printf("Zeichen = %c\n", Zeichen);
15
16    return 0;
17 }
```

4.11. Konvertierung zwischen den Datentypen (Typumwandlung)

Da die Zeichen und der logische Datentyp intern als ganze Zahlen gespeichert werden, liegt es nahe, zwischen den Datentypen konvertieren zu wollen. Dabei muss aber berücksichtigt werden, dass sich nicht jeder Datentyp zu 100% konvertieren lässt; beispielsweise wenn eine reelle Zahl in eine ganze Zahl konvertiert wird, gehen alle Nachkommastellen verloren. Durch die Konvertierung wird die Typkontrolle des Compilers umgangen und kann daher sehr schnell zu Fehlern führen.

Zum Konvertieren gibt es mehrere Möglichkeiten. Hier wird nur die erste Variante vorgestellt (die anderen werden im C++-Teil beschrieben). Dazu wird zuerst der Datentyp, in den konvertiert werden soll, und dahinter der Wert bzw. die Variable geschrieben. Dabei wird der Datentyp in Klammern gesetzt. Diese Konvertierung wird auch **explizite Konvertierung** bzw. **explizite Typumwandlung** genannt.

Beispiel:

```
char c = 'A';
int i;
i = (int) c;
```

Der ASCII-Wert des Buchstaben `'A'` (65) wird als ganze Zahl der Variablen `i` zugeordnet. Auch umgekehrt - bei der Konvertierung von ganzen Zahlen nach Zeichen - wird die ASCII-Tabelle verwendet.

Beispiel:

```
char c;
int i = 97;
c = (char) i; /* ergibt ein 'a' */
```

Um von Ziffernzeichen (`'0'`, `'1'`, ...) nach Ziffern zu konvertieren, wird das Ziffernzeichen in eine ganze Zahl konvertiert und dann der ASCII-Wert des Ziffernzeichens `'0'` abgezogen.

Beispiel:
```
int i;
i = (int) '5' - (int) '0';
```

Bei diesem Beispiel ist der Wert der Variablen `i` gleich 5 (ASCII-Wert von '5' ist 53 und der ASCII-Wert von '0' ist 48). Auch für die Umwandlung von Klein- in Großbuchstaben oder umgekehrt können die ASCII-Werte der Zeichen verwendet werden. Um von Klein- nach Großbuchstaben zu kommen, muss das Zeichen in eine Zahl umgewandelt werden, von dieser Zahl der Wert 32 abgezogen (von Groß- nach Kleinbuchstaben: dazuaddiert) und anschließend wieder in ein Zeichen umgewandelt werden.

Beispiel:
```
char c = 'A';
c = (char) ((int) c + 32); /* Umwandlung in Kleinbuchstaben */
/* oder kurz: */
c = c + 32;
/* oder noch kürzer: */
c += 32;
```

Die kurzen Fassungen sind **implizite Konvertierungen** bzw. **implizite Typumwandlungen**. Dabei wird der Datentyp, in den konvertiert werden soll, weggelassen. Dies kann nur zwischen Zeichen und ganzen Zahlen sowie zwischen logischen Datentypen und ganzen Zahlen angewendet werden. Aber auch zwischen den ganzen Zahlen (z.B. `short` nach `long` oder `long` nach `int`) sowie zwischen den reellen Zahlen (z.B. `double` nach `float`) können implizite Konvertierungen verwendet werden (unter Berücksichtigung der Zahlenbereiche und mit evtl. Genauigkeitsverlust).

Hinweis: Bei der Konvertierung einer ganzen Zahl größer als 255 in ein Zeichen werden die überzähligen Bits nicht berücksichtigt.

5. Einfache Ein- und Ausgabe in C

Ein Programm empfängt einen Strom (englisch: *stream*) von eingegebenen Daten, verarbeitet diese und gibt dann das Ergebnis als Strom von Daten wieder aus. Unter einem Datenstrom versteht man eine Folge von Zeichen. Für einen Datenstrom gibt es verschiedene Standardkanäle, die bereits definiert sind.

Standardkanal	Datenstrom
Standardeingabe (Tastatur)	`stdin`
Standardausgabe (Bildschirm)	`stdout`
Standardfehlerausgabe (Bildschirm)	`stderr`
Standarddrucker (parallele Schnittstelle)	`stdprn`
Standardzusatzgerät (serielle Schnittstelle)	`stdaux`

Diese Standardkanäle können mit Hilfe einiger Funktionen angesprochen werden. Um diese Funktionen sowie deren Verwendung für Standardein- und ausgaben soll es in diesem Kapitel gehen.

Die Standardeingabe geschieht in der Regel über die Tastatur, die Standardausgabe auf dem Bildschirm. Es gibt die Möglichkeit, Ein- und Ausgabe auf Dateien umzulenken, aber darauf wird hier nicht eingegangen.

Die Unterscheidung zwischen der Standardausgabe und der Standardfehlerausgabe dient dem Zweck, zwischen normalen Ausgaben und Fehlermeldungen unterscheiden zu können. Beispielsweise könnten so alle normalen Meldungen auf dem Bildschirm und die Fehlermeldungen in eine Protokolldatei ausgegeben werden.

5.1. Datenausgabe auf den Bildschirm

Mit Hilfe des Bildschirms teilt ein Programm die Ergebnisse von Befehlen dem Benutzer mit. Dabei können konstante und variable Texte und Zahlen formatiert ausgegeben werden. Die Standardfunktion für eine Bildschirmausgabe ist die `printf`-Funktion. Dabei steht `printf` für "print formatted". Um die Funktion verwenden zu können, muss die Headerdatei `stdio.h` mittels `#include <stdio.h>` eingebunden werden.

Die komplette Syntax der `printf`-Funktion lautet:

```
int printf(const char *format [,argument_1 ... ,argument_n]);
```

Dabei haben die einzelnen Wörter die folgenden Bedeutungen:

`int`	Der Datentyp vor dem Funktionsnamen gibt an, was die Funktion als Ergebnis zurückgibt. In diesem Fall ist es eine ganze Zahl (`int`), die angibt, wieviele Zeichen insgesamt ausgegeben wurden. Im Falle eines Fehlers ist das Ergebnis gleich dem Wert `EOF`.
`printf`	der eigentliche Funktionsname
`( ... )`	Innerhalb dieser runden Klammern werden der Funktion die Parameter übergeben.
`const char *format`	Der erste Parameter ist eine Zeichenkette, d.h. eine Folge von Zeichen, die in Anführungsstrichen stehen. In dieser Zeichenkette stehen die Formatierungen für die auszugebenden Daten. Über die Formatierungsangaben wird auch die Anzahl der auszugebenden Daten festgelegt. Die verschiedenen Formatierungsmöglichkeiten sind in den nächsten Tabellen aufgeführt. Mehr zu Zeichenketten finden Sie im Kapitel *Strukturierte Datentypen*.
`argument`	Nach der Formatierungsangabe folgen die Variablennamen, deren Inhalte auf dem Bildschirm ausgegeben werden sollen, jeweils getrennt mit einem Komma. Die Anzahl der Argumente muss mit der Anzahl der Formatierungsanweisungen übereinstimmen.

Die Formatierungszeichenkette beinhaltet zum einen "normale Zeichen", die direkt auf dem Bildschirm ausgegeben werden, und zum anderen die **Formatierungsanweisungen**. Eine Formatierungsanweisung hat folgenden Aufbau:

`%[Flags][Breite][.Präzision][F|N|hh|h|l|ll|L]Typ`

Jede Formatierungsanweisung beginnt mit einem Prozentzeichen. Auf dieses Zeichen folgen:

`[Flags]`	Eine (optionale) Zeichenfolge, über die numerische Vorzeichen, Dezimalpunkte, führende und folgende Nullen, oktale und hexadezimale Präfixe sowie links- und rechtsbündige Ausgabe festgelegt werden.
`[Breite]`	Eine (optionale) Angabe über die minimal auszugebende Zeichenzahl. Notfalls wird mit Leerzeichen oder Nullen aufgefüllt.
`[.Präzision]`	Eine (optionale) Angabe, wieviele Zeichen maximal ausgegeben werden (Zeichenketten), die Minimalzahl von Ziffern (ganze Zahlen) bzw. die maximale Anzahl der Nachkommastellen (Fließkommazahlen).
`[F|N|hh|h|l|ll|L]`	Eine (optionale) Angabe der Größe des Parameters. Genauere Angaben sind in der nächsten Tabelle enthalten.
`Typ`	Die Angabe des Typs der auszugebenden Variable (siehe nächste Tabelle). Diese Angabe muss auf jeden Fall gemacht werden!

Die folgende Tabelle gibt alle möglichen Typen an. Dabei wird erst einmal davon ausgegangen, dass außer der Größenangabe keine optionalen Angaben gemacht werden, d.h. keine Flags, Breite und keine Präzision.

Typ	**optionale Größen-angabe**	**Datentyp des Parameters**	**Ausgabe**	**erlaubte Flags**

`d i`[1]	<keine>
	`hh`[3]
	`h`[1]
	`l`
	`ll`[3]
		`int`
		`char`
		`short`
		`long`
		`long long`[3]
			dezimaler Integer	`- +`
				Leerzeichen

`u`	<keine>
	`hh`[3]
	`h`[1]
	`l`
	`ll`[3]
		`unsigned int`
		`unsigned char`
		`unsigned short`
		`unsigned long`
		`unsigned long long`[3]
			dezimaler Integer
			(nur positive Zahlen)
				`- +`
				Leerzeichen

`o`	<keine>
	`hh`[3]
	`h`[1]
	`l`
	`ll`[3]
		`unsigned int`
		`unsigned char`
		`unsigned short`
		`unsigned long`
		`unsigned long long`[3]
			oktaler Integer
			(nur positive Zahlen)
				`- + #`
				Leerzeichen

`x X`	<keine>
	`hh`[3]
	`h`[1]
	`l`
	`ll`[3]
		`unsigned int`
		`unsigned char`
		`unsigned short`
		`unsigned long`
		`unsigned long long`[3]
			hexadezimaler Integer
			(nur positive Zahlen)
			bei Typ `x`: mit Buchstaben a...f
			bei Typ `X`: mit Buchstaben A...F
				`- + #`
				Leerzeichen

`f F`[4]	<keine> `L`	`float`, `double` `long double`	dezimale Fließkommazahl; vorzeichenbehafteter Wert der Form `[-]d.dddddd`, die Anzahl der Nachkommastellen kann durch die Angabe `.Präzision` festgelegt werden. Ausgabebeispiele: -.900000, 3.141500	`- + #` Leerzeichen
`e E`	<keine> `L`	`float`, `double` `long double`	dezimale Fließkommazahl; vorzeichenbehafteter Wert der Form `[-]d.dddddd e [+|-]dd`	`- + #` Leerzeichen

			(Exponentendarstellung), es steht grundsätzlich eine Ziffer vor dem Dezimalpunkt, die Anzahl der Nachkommastellen kann durch die Angabe `.Präzision` festgelegt werden, der Exponent hat immer zwei Ziffern (notfalls mit führender Null). Ausgabebeispiele: -1.900000e+00, 2.550000e-03	
`g G`	<keine> L	`float`, `double` `long double`	dezimale Fließkommazahl; vorzeichenbehafteter Wert im `e`- oder `f`- bzw. im `E`- oder `F`-Format. Nullen am Ende, ein Dezimalpunkt und ein Vorzeichen werden nur ausgegeben, wenn es notwendig ist. Das `e`-Format wird nur verwendet, wenn das Resultat im `f`-Format mehr als `.Präzision` Stellen ergibt oder mehr als vier führende Nullen erfordert.	`- + #` Leerzeichen
`a`[3] `A`[3]	<keine> L	`float`, `double` `long double`	hexadezimale Fließkommazahl; sonst wie Typ g bzw. G, anstelle des Buchstabens e bzw. E für den Exponenten wird das p bzw. P verwendet, da das e / E zu den hexadezimalen Ziffern gehört.	`- + #` Leerzeichen
`c`	<keine> `l`[2]	`int` `wint_t`	Zeichen	-
`s`	<keine> `l`[3]	`char *` `wchar_t *`	Zeichenkette (Array von `char`) bzw. Zeichenkette (Array von Multibytes) bis zu einem Nullzeichen oder dem Erreichen der durch `.Präzision` vorgegebenen Zeichenzahl.	-
`p`[1]	<keine> `F`[4] `N`[4]	`void *`	Compilerabhängig; Far (F) und Near (N) -Zeiger sind nur in 16-Bit-Compilern enthalten.	Compiler-abhängig
`%`	<keine>	-	Ausgabe des Zeichens `%`	<keine>

[1] erst seit C89; [2] erst seit C95; [3] erst seit C99
[4] gehört nicht zum C-Standard, ist aber in vielen C-Compilern implementiert

Beispiel:

kap05_01.c

```
01 #include <stdio.h>
02
03 int main()
04 {
05    int Zahl = 125;
06    float Reel = 3033.1415f;
07    char Zeichen = 'a';
08
09    printf("Textausgabe ohne Variablen\n");
10    printf("Die Variable Zahl hat den Wert %i\n", Zahl);
11    printf("Variable Zahl in hex. Darstellung: %X\n", Zahl);
12    printf("Die Variable Reel hat den Wert %f\n", Reel);
13    printf("Variable Reel in Exponentendarstellung: %e\n", Reel);
```

```
14     printf("Die Variable Zeichen hat %c als Inhalt.\n", Zeichen);
15
16     return 0;
17 }
```

Dieses Beispielprogramm erzeugt folgende Bildschirmausgabe:

```
Textausgabe ohne Variablen
Die Variable Zahl hat den Wert 125
Variable Zahl in hex. Darstellung: 7D
Die Variable Reel hat den Wert 3033.141602
Variable Reel in Exponentendarstellung: 3.033142e+03
Die Variable Zeichen hat a als Inhalt.
```

Mit der Typangabe alleine können bereits die meisten Bildschirmausgaben bewerkstelligt werden. Für die Feinheiten stehen die optionalen Angaben zur Verfügung:

Flags	**Ausgabe**
`-`	linksbündige Ausgabe (im Normalfall werden Zahlen rechtsbündig ausgegeben)
`+`	numerische Ausgabe immer mit Vorzeichen (im Normalfall wird nur bei negativen Zahlen ein Vorzeichen ausgegeben)
`Leerzeichen`	Positiven Zahlen wird ein Leerzeichen vorangestellt; wird dieses zusammen mit dem + verwendet, wird das Leerzeichen ignoriert!
`#`	Alternative Darstellung Typ `o`: es wird eine `0` vorangestellt Typ `x`/`X`: es wird "`0x`" bzw. "`0X`" vorangestellt Typ `e`/`E`/`f`: es wird ein Dezimalpunkt ausgegeben, auch wenn es keine Nachkommastellen gibt Typ `g`/`G`: wie bei `e` und `E`, zusätzlich werden folgende Nullen nicht unterdrückt
Breite	**Ausgabe**
n	(*n*: Dezimalzahl) Es werden mind. *n* Zeichen ausgegeben und notfalls Leerzeichen vorangestellt.

`0`*n*	(`0`*n*: Dezimalzahl mit vorangestellter `0`) Es werden mind. *n* Zeichen ausgegeben und notfalls Nullen vorangestellt.
	Anstelle einer Zahl kann auch ein Stern ('`*`') eingesetzt werden. Es muss dann aber in der Argumentenliste die Breite als `int` angegeben werden.
	Beispiel:
	`int Breite = 5, Wert = 1;`
	
	`printf("%5d\n", Wert);`
	`printf("%*d\n", Breite, Wert);`

.Präzision	**Ausgabe**
`.0`	Standardvorgabe für ganze Zahlen (Typ `d`, `i`, `u`, `o`, x und `X`); keine Ausgabe von Dezimalpunkt und Nachkommastellen für Fließkommazahlen (Typ `e`, `E`, `f`, `F`, `g`, `G`, `a` und `A`)
`.`*n*	(*n*: Dezimalzahl) Mindestanzahl von auszugebenen Ziffern für ganze Zahlen (Typ `d`, `i`, `u`, `o`, x und `X`); Ausgabe von *n* signifikanten Nachkommastellen für Fließkommazahlen (Typ `e`, `E`, `f` und `F`); Ausgabe von *n* signifikanten Ziffern für Fließkommazahlen (Typ `g`, `G`, `a` und `A`); maximale Anzahl von auszugebenen Zeichen für Zeichenketten (Typ `s`). Auch für die Präzision kann anstelle einer Zahl ein Stern ('`*`') eingesetzt werden. Es muss dann in der Argumentenliste die Präzision als `int` angegeben werden. Wenn für beide, Breite und Präzision, ein Stern eingesetzt wird, muss in der Argumentenliste erst die Angabe der Breite und dann die der Präzision erfolgen. **Beispiel:**

```
int Breite = 5, Praezision = 3;
float Wert = 1.234567;

printf("%5.3f\n", Wert);
printf("%*.*f\n", Breite, Praezision, Wert);
```

Beispiel: Nun wird das obige Beispielprogramm um diese optionalen Angaben erweitert:

kap05_02.c

```
01 #include <stdio.h>
02
03 int main()
04 {
05    int Zahl = 125;
06    float Reel = 3033.1415f;
07    char Zeichen = 'a';
08
09    printf("Textausgabe ohne Variablen\n");
10    printf("Die Variable Zahl hat den Wert %+05i\n", Zahl);
11    printf("Variable Zahl in hex. Darstellung: %#X\n", Zahl);
12    printf("Die Variable Reel hat den Wert %.10f\n", Reel);
13    printf("Variable Reel in Exponentendarst.: %015.3e\n", Reel);
14    printf("Zeichen hat %-3c als Inhalt.\n", Zeichen);
15
16    return 0;
17 }
```

Diesmal wird folgende Bildschirmausgabe erzeugt:

```
Textausgabe ohne Variablen
Die Variable Zahl hat den Wert +0125
Variable Zahl in hex. Darstellung: 0X7D
Die Variable Reel hat den Wert 3033.1416015625
Variable Reel in Exponentendarst.: 0000003.033e+03
Zeichen hat a   als Inhalt.
```

5.2. *Dateneingabe über die Tastatur*

Damit die "Kommunikation" zwischen Computer und Benutzer nicht einseitig aus Bildschirmausgaben besteht, ist es notwendig, dass der Benutzer Eingaben machen kann, die dem Programm (genauer: einer oder mehreren Variablen) übergeben werden. Zur Dateneingabe über die Tastatur wird in erster Linie der Befehl `scanf` verwendet. Die komplette Syntax lautet:

```
int scanf(const char *format [,argument_1 ... ,argument_n]);
```

Der Aufbau der Syntax sowie die einzelnen Parameter für den Befehl sind ganz ähnlich mit denen des `printf`-Befehls. Hier haben die einzelnen Wörter die folgenden Bedeutungen:

`int`	Der Datentyp vor dem Befehl gibt an, was der Befehl als Ergebnis zurückgibt. In diesem Fall ist es eine ganze Zahl (`int`), die angibt, wieviele Werte insgesamt fehlerfrei eingelesen wurden. Im Falle eines Fehlers ist das Ergebnis gleich dem Wert `EOF`.
`scanf`	der eigentliche Befehl
`( ... )`	Innerhalb dieser runden Klammern werden dem Befehl die Parameter übergeben.
`const char *format`	Der erste Parameter ist eine Zeichenkette, d.h. eine Folge von Zeichen, die in Anführungsstrichen stehen. In dieser Zeichenkette stehen die Formatierungen für die einzugebenden Daten. Über die Formatierungs-

	angaben wird auch die Anzahl der einzulesenden Daten festgelegt. Die verschiedenen Formatierungsmöglichkeiten sind in der nächsten Tabelle aufgeführt. Mehr zu Zeichenketten finden Sie im Kapitel *Strukturierte Datentypen*.
`argument`	Nach der Formatierungsangabe folgen die Variablennamen, jeweils getrennt mit einem Komma. Die Variablen müssen allerdings als Zeiger angegeben werden, d.h. es muss ein kaufmännisches Und (`&`) vor jeden Variablennamen gesetzt werden. Mehr zu Zeigern finden Sie im Kapitel *Zeiger*.

Die Formatierungsanweisung ist ganz ähnlich wie bei dem `printf`-Befehl und ist wie folgt aufgebaut:

`%[*][Breite][F|N][hh|h|l|ll|L]Typ`

Auch hier beginnt jede Formatierungsanweisung mit einem Prozentzeichen. Auf dieses Zeichen folgen:

`[*]`	Eine (optionale) Angabe, mit der die Zuweisung zum korrespondierenden Zeiger unterdrückt wird.
`[Breite]`	Eine (optionale) Angabe über die maximal zu lesende Zeichenzahl. Wird vor dem Erreichen der maximalen Zeichenanzahl ein sogenanntes "weißes" Leerzeichen (Trennzeichen wie Leerzeichen, Tabulator, Eingabetaste, ...) eingegeben, wird die Eingabe dieses Wertes beendet und zum nächsten einzugebenden Wert gesprungen.
`[F|N]`	Eine (optionale) Angabe der Größe des Adress-Parameters (`N` = Near-Zeiger, `F` = Far-Zeiger). Diese Angabe ist nur notwendig, wenn die Größe des Parameters von der Standardgröße abweicht.
`[hh|h|l|ll|L]`	Eine (optionale) Angabe der Größe des Parameters. Genauere Angaben sind in der nächsten Tabelle enthalten.
`Typ`	Die Angabe des Typs des einzulesenden Wertes (siehe nächste Tabelle). Diese Angabe muss auf jeden Fall gemacht werden!

Die folgende Tabelle gibt alle möglichen Typen an. Dabei wird erst einmal davon ausgegangen, dass keine optionalen Angaben gemacht werden, d.h. keine Breite usw.

Typ	**optionale Größen-angabe**	**Datentyp des Parameters: Zeiger auf …**	**Eingabe**

`d`	<keine>
	`hh`[3]
	`h`[1]
	`l`
	`ll`[3]
		`int`
		`char`
		`short`
		`long`
		`long long`[3]
			dezimaler Integer

`i`[1]	<keine>
	`hh`[3]
	`h`[1]
	`l`
	`ll`[3]
		`int`
		`char`
		`short`
		`long`
		`long long`[3]
			dezimaler Integer;
			oktaler Interger
			(bei führender 0 und Ziffern 0 … 7);
			hexadezimaler Integer
			(bei führendem 0x bzw. 0X)

`u`	<keine>
	`hh`[3]
	`h`[1]
	`l`
	`ll`[3]
		`unsigned int`
		`unsigned char`
		`unsigned short`
		`unsigned long`
		`unsigned long long`[3]
			dezimaler Integer (nur positive Zahlen)

`o`	<keine>	`unsigned int`	oktaler Integer (unabhängig von führender 0)

	hh[3] h[1] l ll[3]	unsigned char unsigned short unsigned long unsigned long long[3]	
x	<keine> hh[3] h[1] l ll[3]	unsigned int unsigned char unsigned short unsigned long unsigned long long[3]	hexadezimaler Integer (unabhängig von führendem 0x bzw. 0X)
c	<keine> l[1]	char wchar_t	Zeichen (1 Byte) Zeichen (Multibyte)
s	<keine> l[2]	char wchar_t	Zeichenkette (Array von char) Zeichenkette (Array von Multibytes)
p[1]	<kein>	Zeiger auf void	hexadezimaler Integer, der als Speicheradresse interpretiert wird
f e g a[4]	<keine> l L[1]	float double long double	dezimale Fließkommazahl
[	<keine> l[2]	char wchar_t	Zeichenkette (Array von char) Zeichenkette (Array von Multibytes) (siehe Anmerkungen am Ende des Kapitels!)

[1] erst seit C89; [2] erst seit C95; [3] erst seit C99
[4] gehört nicht zum C-Standard, ist aber in vielen C-Compilern implementiert

Die folgenden Angaben gehören wohl nicht zum Standard-C, werden aber von einigen Compilern akzeptiert.

Typ	entspricht Typ	Datentyp des Parameters: Zeiger auf …	Eingabe
D	ld	long	dezimaler Integer
I	li	long	dezimaler Integer
U	lu	unsigned long	dezimaler Integer
O	lo	unsigned long	oktaler Integer
X	lx	unsigned long	hexadezimaler Integer
E	e	float	dezimale Fließkommazahl
G	lg	double	dezimale Fließkommazahl

Beispiel:

kap05_03.c

```
01 #include <stdio.h>
02
03 int main()
04 {
05    int i;
06    float f;
07
08    printf("Bitte eine ganze Zahl & eine reele Zahl eingeben: ");
09    scanf("%i %f", &i, &f);
10    printf("Sie haben %i und %f eingegeben!\n", i, f);
11
12    return 0;
13 }
```

Im Formatstring von `scanf` dürfen **nur** Platzhalter und Leerzeichen verwendet werden!

Nun noch einige Besonderheiten beim Einlesen von Strings:

Bei der Eingabe von Strings kann anstelle des Formats `%s` auch das Format `%[`*`Suchzeichen`*`]` verwendet werden. *Suchzeichen* ist eine Folge von Zeichen, aus denen die Eingabe bestehen muss.

Beispiel: `%[abcd]` erwartet nur Eingaben, die aus den Zeichen `a`, `b`, `c` und `d` bestehen, bei jedem anderen Zeichen ist die Eingabe beendet.

Wird als erstes Suchzeichen das Carret (`^`) verwendet, wird die Eingabe beendet, wenn eines der Suchzeichen eingegeben wird.

Beispiel: `%[^abcd]` erwartet nur Eingaben, die nicht aus den Zeichen `a`, `b`, `c` und `d` bestehen. Die Eingabe wird in diesem Fall auch nicht von den sogenannten "weißen Leerzeichen" beendet, sondern nur durch die angegebenen Suchzeichen oder durch das Erreichen der mit Breite festgelegten Zeichenzahl.

Anstelle von Zeichenaufzählungen können auch Bereiche angegeben werden.

Beispiel: `%[0123456789]` ist äquivalent mit `%[0-9]`.

Bei manchen Compilern kann bzw. muss bei der Formatangabe das `s` angehangen werden (also `%[Suchzeichen]s` anstelle von `%[Suchzeichen]`.

6. Kontrollstrukturen

Mit Kontrollstrukturen sind die Befehle und Zeichenfolgen gemeint, die den Ablauf des Programms steuern. Dazu gehören im wesentlichen verschiedene Verzweigungen und Schleifen.

6.1. Sequenzen

Diese Struktur ist schon bekannt: das Semikolon (";"). Jede Anweisung muss mit einem Semikolon abgeschlossen sein, damit der C-Compiler weiß, an welcher Stelle die Anweisung zu Ende ist.

Es können mehrere Operationen hintereinander gestellt werden, wenn sie durch den Komma-Operator voneinander getrennt werden. Diese Operationen werden von links nach rechts ausgewertet; das Ergebnis der letzten Operation ist gleichzeitig das Ergebnis der gesamten Operation. Diese Eigenschaft wird weiter unten bei der `for`-Schleife ausgenutzt.

Beispiel:

kap06_01.c

```
01 #include <stdio.h>
02
03 int main()
04 {
05    char c_alt, c_neu, c = 'z';
06
07    c_neu = (c_alt = c, c = 'a');
08    printf("c = %c\n", c);
09    printf("c_alt = %c\n", c_alt);
10    printf("c_neu = %c\n", c_neu);
11
12    return 0;
13 }
```

Ausgegeben wird:

```
c = a
c_alt = z
c_neu = a
```

In der Klammer in Zeile 7 wird zuerst der Inhalt von `c` der Variable `c_alt` zugewiesen ('z'). Dann erhält `c` den neuen Wert ('a'). Dieser Wert ist gleichzeitig das Ergebnis des gesamten Klammerausdrucks und wird der Variablen `c_neu` zugewiesen.

Hinweis:

Die Klammer ist dringend notwendig, da der Komma-Operator die geringste Priorität hat. Würde die Klammer weggelassen werden, entspräche es folgender Zeile:

```
   (c_neu = c_alt = c), c = 'a';
```

`c_neu` hätte damit das 'z' und nicht das 'a' als Inhalt.

6.2. Verzweigung: if-Anweisung

Oft ist es nötig, Anweisungen in Abhängigkeit von Bedingungen auszuführen. Der einfachste Befehl zum Abfragen von Bedingungen und anschließendem Verzweigen in Abhängigkeit der Bedingung ist die `if`-Anweisung bzw. `if`-Abfrage (auf **keinen** Fall *`if`-Schleife*!). Sie hat folgende Syntax:

```
if (Bedingung)
   Anweisung1;
```

```
else
   Anweisung2;
```

Gelesen wird dieser Block folgendermaßen: Wenn die *Bedingung* wahr ist, dann führe `Anweisung1` aus, sonst führe `Anweisung2` aus. Dabei kann der `else`-Teil (also `else` `Anweisung2`) weggelassen werden. Wichtig ist, dass nur hinter den Anweisungen ein Semikolon gesetzt wird aber nicht hinter `if (Bedingung)` und `else`.

Beispiel:
kap06_02.c

```
01 /*********************************************
02  * Dieses Programm liest drei ganze Zahlen ein
03  * und ermittelt das Maximum der drei Zahlen.
04  * Eingabe: Zahl1, Zahl2, Zahl3
05  * Ausgabe: Maximum
06  *********************************************/
07 #include <stdio.h>
08
09 int main()
10 {
11    long Zahl1, Zahl2, Zahl3, Maximum;
12
13    printf("Dieses Programm ermittelt das Maximum ");
14    printf("von drei eingegebenen Zahlen.\n");
15    printf("Bitte geben Sie drei ganze Zahlen ein: ");
16    scanf("%ld %ld %ld", &Zahl1, &Zahl2, &Zahl3);
17    if (Zahl1 > Zahl2)
18       Maximum = Zahl1;
19    else
20       Maximum = Zahl2;
21    if (Zahl3 > Maximum)
22       Maximum = Zahl3;
23    printf("Das Maximum der drei Zahlen lautet: ");
24    printf("%li\n", Maximum);
25
26    return 0;
27 }
```

Um anstelle der `Anweisung1` und `Anweisung2` mehrere Anweisungen durchführen zu lassen, müssen diese Anweisungen jeweils in einen Block geschrieben werden, d.h. sie müssen in geschweifte Klammern gesetzt werden. Die Syntax sieht dann folgendermaßen aus:

```
if (Bedingung)
{
   Anweisung1a;
   Anweisung1b;
   Anweisung1c;
}
else
{
   Anweisung2a;
   Anweisung2b;
}
```

`if`-Anweisungen können auch geschachtelt werden. Dabei muss allerdings sehr genau darauf geachtet werden, welches `else` zu welchem `if` gehört. Durch das Einrücken der Anweisungen wird die Zugehörigkeit sehr deutlich sichtbar.

```
if (Bedingung)
{
```

```
   Anweisung1;
   if (Bedingung1)
   {
      Anweisung11a;
      Anweisung11b;
   }
   else
   {
      Anweisung12a;
      Anweisung12b;
   }
}
else
{
   if (Bedingung2)
   {
      Anweisung21a;
      if (Bedingung21)
         Anweisung211;
   }
   else
   {
      Anweisung22a;
      Anweisung22b;
   }
   Anweisung2;
}
```

Beispiel:

kap06_03.c

```
01 /*********************************************
02  * Dieses Programm liest vier ganze Zahlen ein
03  * und ermittelt das Maximum der vier Zahlen.
04  * Eingabe: Zahl1, Zahl2, Zahl3, Zahl4
05  * Ausgabe: Maximum
06  *********************************************/
07 #include <stdio.h>
08
09 int main()
10 {
11    long Zahl1, Zahl2, Zahl3, Zahl4, Maximum;
12
13    printf("Dieses Programm ermittelt das Maximum ");
14    printf("von vier eingegebenen Zahlen.\n");
15    printf("Bitte geben Sie vier ganze Zahlen ein: ");
16    scanf("%ld %ld %ld %ld", &Zahl1, &Zahl2, &Zahl3, &Zahl4);
17    if (Zahl1 > Zahl2)
18       if (Zahl3 > Zahl4)
19          if (Zahl1 > Zahl3)
20             Maximum = Zahl1;
21          else
22             Maximum = Zahl3;
23       else
24          if (Zahl1 > Zahl4)
25             Maximum = Zahl1;
26          else
27             Maximum = Zahl4;
```

```
28    else
29       if (Zahl3 > Zahl4)
30          if (Zahl2 > Zahl3)
31             Maximum = Zahl2;
32          else
33             Maximum = Zahl3;
34       else
35          if (Zahl2 > Zahl4)
36             Maximum = Zahl2;
37          else
38             Maximum = Zahl4;
39    printf("Das Maximum der vier Zahlen lautet: ");
40    printf("%li\n", Maximum);
41
42    return 0;
43 }
```

Eine Mehrfachverzweigung kann als eine Serie von `if-else`-Anweisungen angesehen werden. Dabei ist in jedem (außer im letzten) `else`-Zweig eine weitere `if`-Anweisung enthalten. Dies sieht wie folgt aus:

```
if (Bedingung1)
   Anweisung1;
else
   if (Bedingung2)
      Anweisung2;
   else
      if (Bedingung3)
         Anweisung3;
      else
         Anweisung4;
```

Dabei kann es bei vielen Verzweigung dazu führen, dass der Quelltext sehr weit nach rechts eingerückt wird. Dies führt beim Ausdrucken unter Umständen zu sehr unleserlichen Quelltexten und auch auf dem Bildschirm muss unter Umständen viel nach links und rechts geblättert werden. Daher sollte die Einrückung bei Mehrfachverzweigungen etwas variiert werden:

```
if (Bedingung1)
   Anweisung1;
else if (Bedingung2)
   Anweisung2;
else if (Bedingung3)
   Anweisung3;
else
   Anweisung4;
```

6.3. Verzweigung: ? : -Anweisung

Diese Anweisung ist eine Kurzform der `if`-Anweisung und sieht wie folgt aus:

```
(Bedingung) ? Anweisung1 : Anweisung2;
```

Die entsprechende `if`-Anweisung sieht folgendermaßen aus:

```
if (Bedingung)
   Anweisung1;
else
   Anweisung2;
```

Aber nicht nur Anweisungen sondern auch einfache Ausdrücke (wie Zahlen, Zeichen, usw.) lassen im Gegensatz zur `if`-Anweisung hier einsetzen. Als Ergebnis kann dann der erste oder der zweite Ausdruck

einer Variablen zugeordnet oder in einer Abfrage verwendet werden. Dadurch können kleine Verzweigungen in einer Zeile geschrieben werden. Das nächste Beispiel greift noch einmal die erste Variante der Maximumsbestimmung auf. Jetzt werden aber die `if`-Anweisungen durch `?:`-Anweisungen ersetzt.

Beispiel:

kap06_04.c

```
01 /*********************************************
02  * Dieses Programm liest drei ganze Zahlen ein
03  * und ermittelt das Maximum der drei Zahlen.
04  * Eingabe: Zahl1, Zahl2, Zahl3
05  * Ausgabe: Maximum
06  *********************************************/
07 #include <stdio.h>
08
09 int main()
10 {
11    long Zahl1, Zahl2, Zahl3, Maximum;
12
13    printf("Dieses Programm ermittelt das Maximum ");
14    printf("von drei eingegebenen Zahlen.\n");
15    printf("Bitte geben Sie drei ganze Zahlen ein: ");
16    scanf("%ld %ld %ld", &Zahl1, &Zahl2, &Zahl3);
17    Maximum = (Zahl1 > Zahl2) ? Zahl1 : Zahl2;
18    Maximum = (Zahl3 > Maximum) ? Zahl3 : Maximum;
19    printf("Das Maximum der drei Zahlen lautet: ");
20    printf("%li\n", Maximum);
21
22    return 0;
23 }
```

6.4. Fallunterscheidung: switch-Anweisung

Die `if`-Anweisung ist besonders für Aufgaben geeignet, die nur ein oder zwei Bedingungen betreffen. Durch mehrere gleichwertige Bedingungen (d.h. als Bedingung wird immer wieder die gleiche Variable abgefragt) wird der Programmcode durch die Verschachtelung schnell unübersichtlich. Hierfür gibt es die `switch`-Anweisung. Ihre Syntax sieht wie folgt aus:

```
switch (Ausdruck)
{
   case Wert1:
      Anweisungen1;
      break;
   case Wert2:
      Anweisungen2;
      break;
   case Wert3:
      Anweisungen3;
      break;
   ...
   default:
      Ersatzanweisungen;
}
```

Der Datentyp von Ausdruck muss eine ganze Zahl (`int`) sein oder muss sich in eine ganze Zahl umwandeln lassen, z.B. ein Zeichen (`char`). Reelle Zahlen und Zeichenketten sind nicht zugelassen! Die Werte hinter dem `case` müssen konstante Werte sein. Dabei können die Werte als ganze Zahl oder als Zeichen angegeben werden.

Mit Hilfe des `break`; wird die `switch`-Anweisung beendet. Andernfalls werden die Anweisungen der weiteren `case`-Zeilen auch durchgeführt. Dieses kann auch dazu verwendet werden, um verschiedenen Fälle zusammenzufassen, d.h. in verschiedenen Fällen werden die gleichen Anweisungen ausgeführt (siehe das folgende Beispiel).

Beispiel:

kap06_05.c

```
01 /*********************************************
02  * Dieses Programm fragt das Geschlecht ab und
03  * gibt das Ergebnis auf dem Bildschirm aus.
04  * Eingabe: Geschlecht
05  * Ausgabe: "weiblich" oder "maennlich"
06  *********************************************/
07 #include <stdio.h>
08
09 int main()
10 {
11    char Geschlecht;
12
13    printf("Geben Sie bitte Ihr Geschlecht ein\n");
14    printf("(w: weiblich, m: maennlich): ");
15    scanf("%c", &Geschlecht);
16    switch (Geschlecht)
17    {
18       case 'w':
19       case 'W':
20          printf("weiblich\n");
21          break;
22       case 'm':
23       case 'M':
24          printf("maennlich\n");
25          break;
26       default:
27          printf("keine Angabe\n");
28    }
29
30    return 0;
31 }
```

Neben der `break`-Anweisung kann die `switch`-Anweisung auch mit den Anweisungen `continue` (siehe Abschnitt *break und continue* in diesem Kapitel), `goto` und `return` beendet werden.

Die Reihenfolge, in der der `switch`-Ausdruck mit den `case`-Ausdrücken verglichen wird, ist nicht definiert. Auch die Umsetzung der Vergleiche in die Maschinensprache hängt von der Anzahl und den Werten der `case`-Ausdrücke ab. D.h., die `switch`-Anweisung kann nicht als Folge von `if`-Anweisungen angesehen werden!

Die Anzahl der `case`-Ausdrücke, die in einer `switch`-Anweisung verwendet werden dürfen, ist begrenzt. In C89 dürfen maximal 257, in C99 maximal 1023 `case`-Ausdrücke verwendet werden.

Obwohl für den Datentypen des Ausdrucks alle Typen von ganzen Zahlen zugelassen sind, gibt es einige ältere Compiler, die die Datentypen `long` und `unsigned long` für den Ausdruck nicht zulassen.

6.5. *Schleifen: while-Schleifen*

Soll ein Programmteil mehrmals wiederholt werden, wird eine sogenannte **Schleife** verwendet. In C gibt es drei verschiedene Schleifen: `while`-, `do-while`- und `for`-Schleifen. Die beiden letzten lassen sich auch durch die `while`-Schleife darstellen.

Die `while`-Schleife hat folgende Syntax:

```
while (Bedingung)
   Anweisung;
```

Ähnlich wie bei der `if`-Anweisung sollte die *Anweisung* eingerückt werden, um schneller zu erkennen, an welcher Stelle die `while`-Schleife zu Ende ist. Sollen mehrere Anweisungen in einer Schleife wiederholt werden, werden diese (wie bei der `if`-Anweisung) in einen Block geschrieben. Dies sieht dann wie folgt aus:

```
while (Bedingung)
{
   Anweisung1;
   Anweisung2;
   Anweisung3;
}
```

Dabei wird die *Anweisung* bzw. der Anweisungsblock so lange wiederholt ausgeführt, bis die *Bedingung* nicht mehr wahr ist. Ist die *Bedingung* von vornherein nicht erfüllt, wird die *Anweisung* bzw. der Anweisungsblock in der `while`-Schleife überhaupt nicht ausgeführt. Anders herum: Wird die *Bedingung* niemals falsch, wird die Schleife auch niemals beendet. Dies wird eine **Endlosschleife** genannt.

Beispiel:

kap06_06.c

```
01 /*****************************************
02  * Dieses Programm berechnet die Fakultaet
03  * einer eingegebenen Zahl und gibt das
04  * Ergebnis auf dem Bildschirm aus.
05  * Eingabe: Zahl
06  * Ausgabe: Fakult
07  *****************************************/
08 #include <stdio.h>
09
10 int main()
11 {
12    unsigned int Zahl, Zaehler = 1;
13    unsigned long Fakult = 1;
14
15    printf("Dieses Programm berechnet die ");
16    printf("Fakultaet einer ganzen Zahl.\n");
17    printf("Geben Sie bitte eine ganze Zahl ein: ");
18    scanf("%d", &Zahl);
19    while (Zaehler <= Zahl)
20    {
21       Fakult = Fakult * Zaehler;
22       Zaehler++;
23    }
24    printf("%u! = %lu\n", Zahl, Fakult);
25
26    return 0;
27 }
```

Mit Hilfe der Kurzform-Operatoren und der Inkrementierung von Variablen kann die `while`-Schleife drastisch verkürzt werden:

```
while (Zaehler <= Zahl)
   Fakult *= Zaehler++;
```

Genauso wie `if`-Anweisungen lassen sich `while`-Schleifen schachteln. Auch hierbei hilft das Einrücken der Anweisungen innerhalb der Schleife, um den Überblick über die jeweiligen Schleifenenden zu haben. Im folgenden Beispiel wird um die Schleife der Fakultätsberechnung eine weitere Schleife herumgelegt.

Beispiel:

kap06_07.c

```
01 /*****************************************
02  * Dieses Programm berechnet die Fakultaet
03  * einer eingegebenen Zahl und gibt das
04  * Ergebnis auf dem Bildschirm aus.
05  * Eingabe: Zahl
06  * Ausgabe: Fakult
07  *****************************************/
08 #include <stdio.h>
09
10 int main()
11 {
12    unsigned int Zahl, Zaehler;
13    unsigned long Fakult;
14    char c = 'j';
15
16    printf("Dieses Programm berechnet die ");
17    printf("Fakultaet einer ganzen Zahl.\n");
18    while (c == 'j' || c == 'J')
19    {
20       printf("Geben Sie bitte eine ganze Zahl ein: ");
21       scanf("%d", &Zahl);
22
23       Zaehler = 1;
24       Fakult = 1;
25       while (Zaehler <= Zahl)
26          Fakult *= Zaehler++;
27       printf("%u! = %lu\n", Zahl, Fakult);
28       printf("Moechten Sie eine weitere Zahl ");
29       printf("berechnen? (j/n) ");
30       c = 0;
31       while (c != 'j' && c != 'J' && c != 'n' && c != 'N')
32          scanf("%c", &c);
33    }
34
35    return 0;
36 }
```

Im Gegensatz zum vorigen Beispielprogramm ist zu beachten, dass es jetzt nicht mehr ausreicht, die Variablen `Zaehler` und `Fakult` einmalig mit `1` zu initialisieren. Da die Berechnung mehrmals durchgeführt werden kann, ist es wichtig, diese beiden Variablen direkt vor der Berechnung auf den Startwert zu setzen.

Bei der äußeren `while`-Schleife ist es von Nachteil, dass die Variable `c` zuvor mit dem Zeichen `'j'` initialisiert sein muss. Dies ist bei der `do-while`-Schleife anders.

6.6. *Schleifen: do-while-Schleifen*

Bei der `do-while`-Schleife ist die Abfrage der *Bedingung* am Ende. Dadurch wird die *Anweisung* bzw. der Anweisungsblock innerhalb der Schleife auf jeden Fall mindestens einmal ausgeführt (Motto: Erst zuschlagen, dann fragen), während bei der `while`-Schleife die *Anweisung* bzw. der Anweisungsblock bei falscher *Bedingung* überhaupt nicht ausgeführt wird. Die Syntax für die `do-while`-Schleife sieht folgendermaßen aus:

```
do
   Anweisung;
while (Bedingung);
```

Entsprechend die Syntax mit einem Anweisungsblock:

```
do
{
   Anweisung1;
   Anweisung2;
   Anweisung3;
} while (Bedingung);
```

Wichtig: Im Gegensatz zur `while`-Schleife wird hier hinter `while (Bedingung);` ein Semikolon gesetzt! Im folgenden Beispiel werden zwei der drei `while`-Schleifen vom vorigen Beispiel durch `do-while`-Schleifen ersetzt. Dadurch fällt die Initialisierung der Variablen in der *Bedingung* weg.

Beispiel:

kap06_08.c

```
01 /*******************************************
02  * Dieses Programm berechnet die Fakultaet
03  * einer eingegebenen Zahl und gibt das
04  * Ergebnis auf dem Bildschirm aus.
05  * Eingabe: Zahl
06  * Ausgabe: Fakult
07  *******************************************/
08 #include <stdio.h>
09
10 int main()
11 {
12    unsigned int Zahl, Zaehler;
13    unsigned long Fakult;
14    char c;
15
16   printf("Dieses Programm berechnet die ");
17   printf("Fakultaet einer ganzen Zahl.\n");
18   do
19   {
20      printf("Geben Sie bitte eine ganze Zahl ein: ");
21      scanf("%d", &Zahl);
22
23       Zaehler = 1;
24       Fakult = 1;
25       while (Zaehler <= Zahl)
26          Fakult *= Zaehler++;
27       printf("%u! = %lu\n", Zahl, Fakult);
28       printf("Moechten Sie eine weitere Zahl ");
29       printf("berechnen? (j/n) ");
30       do
31          scanf("%c", &c);
32       while (c != 'j' && c != 'J' && c != 'n' && c != 'N');
```

```
33     } while (c == 'j' || c == 'J');
34
35     return 0;
36 }
```

6.7. Schleifen: for-Schleifen

Die dritte Schleife ist die `for`-Schleife und ist eine sogenannte **Zählschleife**. Der Name kommt daher, dass ursprünglich bei dieser Schleifenform mit Hilfe einer Zahlenvariablen mitgezählt wurde, wie oft die Schleife ausgeführt werden soll. Entsprechend wird in der *Bedingung* die Zahlenvariable abgefragt, ob die gewünschte Anzahl von Schleifendurchläufen bereits erreicht ist. In C/C++ sieht die `for`-Schleife etwas anders aus als in anderen Programmiersprachen. Dadurch ist es möglich, auch `for`-Schleifen ohne Zähler zu konstruieren.

Die Syntax sieht folgendermaßen aus:

```
for ([Variablen-Initialisierung];[Bedingung];[Veränderung])
   Anweisung;
```

bzw. mit mehreren Anweisungen in einem Anweisungsblock:

```
for ([Variablen-Initialisierung];[Bedingung];[Veränderung])
{
   Anweisung1;
   Anweisung2;
   Anweisung3;
}
```

Das folgende Beispiel gibt alle Buchstaben von 'A' bis 'Z' auf dem Bildschirm aus. In diesem Fall ist die Zählvariable die `char`-Variable `c`.

Beispiel:

kap06_09.c

```
01 /*****************************************
02  * Dieses Programm gibt alle Buchstaben von
03  * 'A' bis 'Z' auf dem Bildschirm aus.
04  *****************************************/
05 #include <stdio.h>
06
07 int main()
08 {
09    char c;
10
11    printf("Dieses Programm gibt alle Buchstaben ");
12    printf("von 'A' bis 'Z' aus.\n");
13    for (c = 'A'; c <= 'Z'; c++)
14       printf("%c ", c);
15    printf("\n");
16
17    return 0;
18 }
```

In der Syntax sind die *Variablen-Initialisierung*, die *Bedingung* und die *Veränderung* jeweils in eckigen Klammern angegeben, d.h. sie sind optional. So kann zum Beispiel die *Variablen-Initialisierung* entfallen, wenn die Variablen bereits vorher initialisiert wurden. Die *Veränderung* des Zählers kann entfallen, z.B. wenn er innerhalb der Schleife verändert wird. Auch die *Bedingung* kann weggelassen werden, allerdings ist dann die Schleife eine Endlosschleife! Im folgenden Beispiel werden *Variablen-Initialisierung* und *Veränderung* weggelassen (oder besser gesagt: an andere Stellen umgesetzt). Trotzdem müssen die Semikolons gesetzt werden.

Beispiel:

kap06_10.c

```
01 /*****************************************
02  * Dieses Programm gibt alle Buchstaben von
03  * 'A' bis 'Z' auf dem Bildschirm aus.
04  *****************************************/
05 #include <stdio.h>
06
07 int main()
08 {
09    char c = 'A'; /* Variablen-Initialisierung */
10
11    printf("\nDieses Programm gibt alle Buchstaben ");
12    printf("von 'A' bis 'Z' aus.\n");
13    for ( ; c <= 'Z'; )
14       printf("%c ", c++);
15    printf("\n");
16
17    return 0;
18 }
```

Für *Variablen-Initialisierung*, *Bedingung* und *Veränderung* können aber auch mehrere Einträge gemacht werden. Diese müssen mit Kommata voneinander getrennt werden (siehe Abschnitt *Sequenzen* am Anfang dieses Kapitels). Werden mehrere Befehle oder Abfragen als Bedingung angegeben, muss darauf geachtet werden, dass nur das Ergebnis des letzten Befehls bzw. der letzten Abfrage für die Bedingung verwendet wird. Im folgenden Beispiel wird eine Berechnung innerhalb einer `for`-Schleife in mehreren Schritten so weit verkürzt, dass keine Anweisung innerhalb der Schleife mehr übrigbleibt. In diesem Fall muss ein Semikolon (sozusagen eine Leeranweisung) als Anweisung für die `for`-Schleife gesetzt werden.

Beispiel:

kap06_11.c

```
01 #include <stdio.h>
02
03 int main()
04 {
05    unsigned long a, b = 1, c = 1;
06    /* a wird in der for-Schleife initialisiert */
07
08    printf("         a |          b |          c\n");
09    printf("-----------------------------------\n");
10    for (a = 1; a < 10; a = a + 1)
11    {
12       b = b * c;
13       c = c + a;
14       printf("%10lu | %10lu | %10lu\n", a, b, c);
15       c = c + 1;
16    }
17    printf("-----------------------------------\n");
18
19    return 0;
20 }
```

Im ersten Schritt beim Verkürzen der `for`-Schleife werden die Kurzform-Operatoren eingesetzt. Ferner werden auch die Startwerte der Variablen `b` und `c` in der `for`-Schleife gesetzt.

kap06_12.c

```c
#include <stdio.h>

int main()
{
   unsigned long a, b, c;

   printf("         a |          b |          c\n");
   printf("-----------------------------------\n");
   for (a = 1, b = 1, c = 1; a < 10; a++)
   {
      b *= c;
      c += a;
      printf("%10lu | %10lu | %10lu\n", a, b, c);
      c++;
   }
   printf("-----------------------------------\n");

   return 0;
}
```

Im zweiten Schritt werden die Initialisierungen zu einer zusammengefasst. Außerdem wird das `c++` zu den *Veränderungen* verschoben.

kap06_13.c

```c
#include <stdio.h>

int main()
{
   unsigned long a, b, c;

   printf("         a |          b |          c\n");
   printf("-----------------------------------\n");
   for (a = b = c = 1; a < 10; a++, c++)
   {
      b *= c;
      c += a;
      printf("%10lu | %10lu | %10lu\n", a, b, c);
   }
   printf("-----------------------------------\n");

   return 0;
}
```

Im letzten Schritt werden die drei Anweisungen in die *Bedingung* verschoben. Wichtig ist, dass die ursprüngliche *Bedingung* die letzte ist. Nun werden aber vor der Abfrage der *Bedingung* die drei Anweisungen ausgeführt. Der Ablauf ist damit ähnlich wie bei der `do-while`-Schleife. Deshalb muss die Anzahl der Schleifendurchgänge um eins verringert werden.

kap06_14.c

```c
#include <stdio.h>

int main()
{
   unsigned long a, b, c;

   printf("         a |          b |          c\n");
   printf("-----------------------------------\n");
   for (a = b = c = 1; b *= c, c += a, printf("%10lu | %10lu | "
                           "%10lu\n", a, b, c), a < 9; a++, c++)
```

```
11        ;
12     printf("---------------------------------------\n");
13
14     return 0;
15  }
```

Natürlich ist diese komprimierte Schreibweise nicht sehr leserlich, sehr fehleranfällig und daher auch nicht zu empfehlen!

Genau wie bei den beiden anderen Schleifenformen können auch `for`-Schleifen beliebig verschachtelt werden. Zur besseren Übersicht sollten auch hier die Anweisungen innerhalb der Schleife eingerückt werden.

6.8. *break und continue*

Mit den Befehlen `break` und `continue` gibt es noch zwei Befehle, mit denen der Ablauf innerhalb von Schleifen beeinflusst werden kann. Der `break`-Befehl wurde bereits in der `switch`-Anweisung verwendet, um aus dieser herauszuspringen. Genauso ist es auch bei den Schleifen: Mit Hilfe des `break`-Befehls wird die Schleife unabhängig von der *Bedingung* beendet und mit dem nächsten Befehl nach der Schleife fortgefahren. Dies gilt für alle drei Schleifenformen. Als Beispiel dient noch einmal das Programm zur Berechnung der Fakultät.

Beispiel:

kap06_15.c

```
01 /*****************************************
02  * Dieses Programm berechnet die Fakultaet
03  * einer eingegebenen Zahl und gibt das
04  * Ergebnis auf dem Bildschirm aus.
05  * Eingabe: Zahl
06  * Ausgabe: Fakult
07  *****************************************/
08 #include <stdio.h>
09
10 int main()
11 {
12    unsigned int Zahl, Zaehler;
13    unsigned long Fakult;
14    char c;
15
16    printf("Dieses Programm berechnet die ");
17    printf("Fakultaet einer ganzen Zahl.\n");
18    while (1) /* 1 => Wahr => Endlosschleife! */
19    {
20       printf("Geben Sie bitte eine ganze Zahl ein: ");
21       scanf("%d", &Zahl);
22
23       Zaehler = 1;
24       Fakult = 1;
25       while (Zaehler <= Zahl)
26          Fakult *= Zaehler++;
27       printf("%u! = %lu\n", Zahl, Fakult);
28       printf("Moechten Sie eine weitere Zahl ");
29       printf("berechnen? (j/n) ");
30       do
31          scanf("%c", &c);
32       while (c != 'j' && c != 'J' && c != 'n' && c != 'N');
33       if (c == 'n' || c == 'N')
34          break; /* hier wird die Schleife verlassen */
```

```
35     }
36
37     return 0;
38 }
```

Der `continue`-Befehl ist mehr oder weniger das Gegenteil des `break`-Befehls: Mit ihm wird zum Ende des Schleifenkörpers gesprungen. Bei `while`- und `do-while`-Schleifen wird nach `continue` die *Bedingung* abgefragt, bei `for`-Schleifen wird nach `continue` erst die *Veränderung* aufgerufen und dann die *Bedingung* abgefragt.

Als Beispiel wird das Programm zur Berechnung der Fakultät erweitert: Nach der Benutzereingabe der ganzen Zahl wird diese geprüft, ob sie größer oder gleich eins ist. Wenn nicht, wird mit `continue` die Schleife neu gestartet.

Beispiel:

kap06_16.c

```
01 /*****************************************
02  * Dieses Programm berechnet die Fakultaet
03  * einer eingegebenen Zahl und gibt das
04  * Ergebnis auf dem Bildschirm aus.
05  * Eingabe: Zahl
06  * Ausgabe: Fakult
07  *****************************************/
08 #include <stdio.h>
09
10 int main()
11 {
12    int Zahl, Zaehler;
13    unsigned long Fakult;
14    char c;
15
16    printf("Dieses Programm berechnet die ");
17    printf("Fakultaet einer ganzen Zahl.\n");
18    while (1) /* 1 => Wahr => Endlosschleife! */
19    {
20       printf("Geben Sie bitte eine ganze Zahl ein: ");
21       scanf("%d", &Zahl);
22       if (Zahl < 1)
23          continue;   /* Schleife von vorne beginnen */
24       Zaehler = 1;
25       Fakult = 1;
26       while (Zaehler <= Zahl)
27          Fakult *= Zaehler++;
28       printf("%u! = %lu\n", Zahl, Fakult);
29       printf("Moechten Sie eine weitere Zahl ");
30       printf("berechnen? (j/n) ");
31       do
32          scanf("%c", &c);
33       while (c != 'j' && c != 'J' && c != 'n' && c != 'N');
34       if (c == 'n' || c == 'N')
35          break;   /* hier wird die Schleife verlassen */
36    }
37
38    return 0;
39 }
```

6.9. *goto*

So wie im guten alten Basic gibt es auch in C/C++ den `goto`-Befehl. Im Normalfall kommt man ohne diesen Befehl aus, aber es gibt ja immer die berühmten Ausnahmen. Generell werden Programme mit `goto`-Befehlen sehr schnell unübersichtlich!

Die Syntax lautet:

`goto` *`Labelname`*`;`

Dabei ist *Labelname* ein beliebiger Name, der irgendwo im Programm (zumindest in der gleichen Funktion!) definiert sein muss. Ein Label wird ganz einfach definiert. Dazu wird der Labelname gefolgt von einem Doppelpunkt vor den Befehl bzw. vor die Anweisung geschrieben, zu der der `goto`-Befehl springen soll. Damit diese Labels später schneller wiedergefunden werden, sollten sie direkt an den Anfang der Zeile geschrieben werden. Der Befehl dahinter bleibt wie bisher eingerückt.

Als Beispiel wird das vorige Programm genommen. Hier wird der `continue`-Befehl durch den `goto`-Befehl ersetzt. Das Label wird vor der ersten Anweisung in der `while`-Schleife geschrieben.

Beispiel:

kap06_17.c

```
01 /*****************************************
02  * Dieses Programm berechnet die Fakultaet
03  * einer eingegebenen Zahl und gibt das
04  * Ergebnis auf dem Bildschirm aus.
05  * Eingabe: Zahl
06  * Ausgabe: Fakult
07  *****************************************/
08 #include <stdio.h>
09
10 int main()
11 {
12    int Zahl, Zaehler;
13    unsigned long Fakult;
14    char c;
15
16    printf("Dieses Programm berechnet die ");
17    printf("Fakultaet einer ganzen Zahl.\n");
18    while (1) /* 1 => Wahr => Endlosschleife! */
19    {
20 neu:  printf("Geben Sie bitte eine ganze Zahl ein: ");
21       scanf("%d", &Zahl);
22       if (Zahl < 1)
23          goto neu;   /* Schleife von vorne beginnen */
24       Zaehler = 1;
25       Fakult = 1;
26       while (Zaehler <= Zahl)
27          Fakult *= Zaehler++;
28       printf("%u! = %lu\n", Zahl, Fakult);
29       printf("Moechten Sie eine weitere Zahl ");
30       printf("berechnen? (j/n) ");
31       do
32          scanf("%c", &c);
33       while (c != 'j' && c != 'J' && c != 'n' && c != 'N');
34       if (c == 'n' || c == 'N')
35          break;   /* hier wird die Schleife verlassen */
36    }
37
```

```
38     return 0;
39 }
```

Ein sehr gefährliches Feature der `goto`-Anweisung ist die Möglichkeit, mitten in Blöcke springen zu können. Dabei wird auch die Initialisierung von Variablen, die am Blockanfang definiert werden, übersprungen (zumindest von den meisten Compilern). Das Ergebnis ist, dass der Inhalt der Variablen nicht vorhersehbar ist.

Beispiel:

kap06_18.c

```
01 #include <stdio.h>
02
03 int main()
04 {
05    goto Label;
06
07    {
08       int Zahl = 10;
09
10 Label:
11       printf("Zahl = %i\n", Zahl); /* irgendeine Zahl wird ausgegeben! */
12    }
13
14    return 0
15 }
```

7. Strukturierte Datentypen

7.1. Arrays

Ein **Array** ist eine Sammlung bzw. eine Tabelle mit Variablen des gleichen Datentyps, die über einen gemeinsamen Variablennamen angesprochen werden. Jede Variable des Arrays ist von den anderen unabhängig und wird als **Element** bezeichnet. Der Datentyp ist dabei beliebig, er kann auch selber ein strukturierter Datentyp sein, z.B. ein Array von Arrays. Innerhalb des Arrays wird mit Hilfe einer Positionsangabe, dem **Index**, auf ein Element zugegriffen. **Der Index zählt immer ab Null!**

Ein Array wird folgendermaßen definiert:

Datentyp Arrayname[Anzahl_der_Elemente]

Beispiel:
```
int Zahlen[10];
```

Im Beispiel wird ein Array von ganzen Zahlen (Integer) namens `Zahlen` mit 10 Elementen angelegt. Die einzelnen Elemente werden mit `Zahlen[0]` ... `Zahlen[9]` angesprochen. Das Element `Zahlen[10]` gibt es nicht, da es bereits das 11. Element wäre. Allerdings wird weder vom Compiler noch zur Laufzeit überprüft, ob die Array-Grenzen überschritten werden. Daher muss der Programmierer immer selber sicherstellen, dass dies nicht geschieht.

Arrays werden meist mit Schleifen behandelt, weil häufig für alle Elemente des Arrays die gleichen Operationen durchgeführt werden müssen. Im folgenden Beispiel werden 10 Zahlen in ein Array eingelesen und anschließend die Summe der 10 Zahlen ausgegeben.

Beispiel:

kap07_01.c

```
01 /*********************************************
02  * Dieses Programm liest zehn ganze Zahlen ein
03  * und ermittelt die Summe der zehn Zahlen.
04  * Eingabe: Zahlen[0], ..., Zahlen[9]
05  * Ausgabe: Summe
06  *********************************************/
07 #include <stdio.h>
08
09 int main()
10 {
11    int Zahlen[10], i, Summe;
12
13    printf("Dieses Programm ermittelt die Summe ");
14    printf("von zehn eingegebenen Zahlen.\n");
15    printf("Bitte geben Sie zehn ganze Zahlen ein: ");
16    for (i = 0; i < 10; i++)
17       scanf("%d", &Zahlen[i]);
18    for (i = 0, Summe = 0; i < 10; i++)
19    {
20       printf("%i", Zahlen[i]);
21       printf(i < 9 ? " + " : " = ");
22       Summe += Zahlen[i];
23    }
24    printf("%i\n", Summe);
25
26    return 0;
27 }
```

Die Gesamtgröße des Arrays (d.h. der für das Array benötigte Speicherplatz) ergibt sich aus der Anzahl der Elemente multipliziert mit der Größe eines Elementes. Um die Größe eines Elementes zu erhalten, kann der Befehl `sizeof(`*`Datentyp`*`)` bzw. `sizeof(`*`Variable`*`)` verwendet werden.

Beispiel:
```
printf("Größe des Arrays: %i", 10 * sizeof(int));      /* bzw. */
printf("Größe des Arrays: %i", 10 * sizeof(Zahlen[0]));
```

Daraus ergibt sich eine Arraygröße von 40 Bytes (ein `int` hat 4 Bytes).

Arrays lassen sich genauso wie alle anderen Variablen bei der Definition auch gleich initialisieren. Dazu werden alle Werte für das Array jeweils mit einem Komma getrennt in geschweifte Klammern gestellt. Das sieht dann so aus:

Beispiel:
```
int Zahlen[10] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
```

Die Anzahl der Elemente des Arrays und die Anzahl der Initialisierungswerte muss übereinstimmen. Da der Computer mitzählt, wieviele Elemente initialisiert werden, kann bei der Initialisierung die Anzahl der Elemente in den eckigen Klammern entfallen, also:

```
int Zahlen[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
```

Es lassen sich auch mehrdimensionale Arrays definieren. Dabei muss für jede Dimension die Anzahl der Elemente in eckigen Klammern angegeben werden, und zwar so, dass jede Dimension ein eigenes Klammerpaar hat, z.B. `Matrix[3][3]`. Dagegen wäre `Matrix[3,3]` falsch (ist identisch mit `Matrix[3]`).

Beispiel:

kap07_02.c

```
01 /*********************************************
02  * Dieses Programm gibt die Zahlen einer
03  * 3x3-Matrix auf dem Bildschirm aus.
04  * Eingabe: keine
05  * Ausgabe: alle Werte der 3x3-Matrix
06  *********************************************/
07 #include <stdio.h>
08
09 int main()
10 {
11    int i, j, Matrix[3][3] = { {1, 2, 3}, {4, 5, 6}, {7, 8, 9} };
12
13    /* Ausgabe der 3*3-Matrix:*/
14    for (i = 0; i < 3; i++)
15    {
16       for (j = 0; j < 3; j++)
17          printf("%3i ", Matrix[i][j]);
18       printf("\n"); /* Zeilenumbruch nach jeder Matrixzeile */
19    }
20
21    return 0;
22 }
```

Mit dem C99-Standard ist die Möglichkeit hinzugekommen, Arrays mit einer variablen Anzahl von Elementen anzulegen. D.h. zur Zeit des Compilierens ist die Anzahl der Elemente des Arrays noch nicht bekannt. Ein solches Array wird wie die bisherigen Arrays definiert, nur die Anzahl der Elemente ist ein nicht-konstanter Ausdruck. Zur Laufzeit des Programms wird bei der Definition des Arrays der nicht-konstante Ausdruck (muss eine positive ganze Zahl sein!) ausgewertet und dann das Array mit der ermittelten Elementanzahl angelegt. Ist das Array erst einmal definiert, kann die Anzahl der Elemente nicht mehr geändert werden; das Array ist also kein dynamisches Array!

Beispiel:

kap07_03.c

```
01 #include <stdio.h>
02
03 int main()
04 {
05    int Anzahl = 0, i;
06
07    printf("Wie viele Elemente? ");
08    scanf("%i", &Anzahl);
09
10    if (Anzahl > 0)
11    {
12       int Array[Anzahl]; /* funktioniert erst ab C99! */
13
14       for (i = 0; i < Anzahl; i++)
15          Array[i] = i + 1;
16       for (i = 0; i < Anzahl; i++)
17          printf("Array[%i] = %i\n", i, Array[i]);
18    }
19
20    return 0;
21 }
```

Arrays mit variabler Anzahl von Elementen dürfen nicht als `static`- oder `extern`-Variablen angelegt werden; auch dürfen sie nicht als globale Variablen angelegt werden.

Wird ein neuer Datenyp mittels `typedef` anhand eines Arrays mit variabler Elementanzahl definiert, wird die Anzahl der Elemente für das Array beim Abarbeiten des `typedef` ermittelt. Der so definierte Datentyp ändert sich dann nicht mehr!

Beispiel:

kap07_04.c

```
#include <stdio.h>

int main()
{
   int Anzahl = 5;

   typedef int Array[Anzahl];

   Anzahl++;

   Array A;         /* Array mit 5 Elementen */
   int B[Anzahl];   /* Array mit 6 Elementen */

   return 0;
}
```

Im Kapitel *Zeiger* wird noch einmal auf Arrays eingegangen.

7.2. Zeichenketten (Strings)

Eine **Zeichenkette** (auch **String** genannt) ist ein Spezialfall eines Arrays, nämlich ein eindimensionales Array von Zeichen (`char`). In jedem Element des Arrays wird genau ein Zeichen abgelegt. Alle Elemente

zusammen ergeben die Zeichenkette, z.B. ein Satz. Damit eignen sich Zeichenketten zum Speichern von Texten, Meldungen, usw. Dabei kann mit dem Array auf jeden Buchstaben einzeln zugegriffen werden.

Damit im Programm das Textende eindeutig erkannt werden kann, wird nach dem letzten Zeichen der Zeichenkette noch ein weiteres Element mit dem Zeichen '\0' (ASCII-Wert: 0) gespeichert. Das Array muss also immer ein Element mehr haben als die Zeichenkette lang ist!!! Wird dieses nicht beachtet, wird das Programm fehlerhaft arbeiten oder sogar abstürzen!

Natürlich können Zeichenketten wie gewöhnliche Arrays bei der Definition initialisiert werden. Das würde dann so aussehen:

```
char Text[] = {'D','i','e','s',' ','i','s','t',' ',
               'e','i','n',' ','T','e','x','t','!','\0'};
```

Damit wird ein Array mit 19 Elementen (18 Textzeichen und 1 Abschlusszeichen) erzeugt. Tabellarisch sieht das folgendermaßen aus:

D	i	e	s		i	s	t		e	i	n		T	e	x	t	!	\0

Damit die Initialisierungen von Zeichenketten nicht immer buchstabenweise erfolgen müssen, werden die Texte einfach in Anführungsstriche gesetzt. Das obige Array lässt sich dadurch einfacher initialisieren:

```
char Text[] = "Dies ist ein Text!";
```

Durch die Anführungsstriche setzt der Compiler automatisch das Abschlusszeichen an den Text heran; der Programmierer braucht sich an dieser Stelle darum nicht mehr zu kümmern.

Gefährlich wird es, wenn eine Zeichenkette zum Einlesen von Texten benutzt wird. Genau wie bei den Arrays wird weder vom Compiler noch zur Laufzeit getestet, ob die Zeichenkette für den eingegebenen Text groß genug ist! Wird ein Text über die Größe des Arrays hinweg eingelesen, werden im Normalfall die im Speicher hinter dem Array abgelegten Variablen überschrieben (das Programm liefert dann u.U. falsche Ergebnisse!); im ungünstigsten Fall stürzt das Programm ab (Speicherzugriffsfehler)! Zum Einlesen von Zeichenketten sollten also die Arrays ausreichend dimensioniert werden. Ferner sollte beim `scanf` die maximale Anzahl von erlaubten Zeichen angegeben werden. Alle weiteren eingegebenen Zeichen bleiben im Tastaturpuffer.

Beispiel:

kap07_05.c

```
01 /*********************************************
02  * Dieses Programm liest Ihren Namen ein und
03  * gibt ihn wieder auf dem Bildschirm aus.
04  *********************************************/
05 #include <stdio.h>
06
07 int main()
08 {
09    char Name[100];       /* Name darf max. 99 Zeichen lang */
10                          /* sein + 1 Abschlusszeichen */
11
12    printf("Geben Sie bitte Ihren Namen ein: ");
13    scanf("%99s", Name); /* KEIN & vor dem Variablennamen!!! */
14    printf("So, so, Sie heissen also %s\n", Name);
15
16    return 0;
17 }
```

Dadurch, dass die Eingabe nach jedem "weißen Leerzeichen" endet, kann auf diese Art und Weise nicht Vor- und Nachname zusammen eingegeben werden. In dem Array wird nur der Vorname bis zum ersten Leerzeichen übernommen. Darauf wird hier aber nicht weiter eingegangen.

Eine Zeichenkette lässt sich nicht direkt einer anderen Zeichenkette zuweisen, sondern muss zeichenweise kopiert werden. Dies wird im folgenden Beispiel gezeigt.

Beispiel:

kap07_06.c

```
01 /*****************************************
02  * Dieses Programm kopiert einen Text einer
03  * Zeichenkette in eine andere.
04  *****************************************/
05 #include <stdio.h>
06
07 int main()
08 {
09    char Kette1[] = "Ein Text!", Kette2[100];
10    int i = 0;
11
12    printf("Der Text von Kette1 (%s) ", Kette1);
13    printf("wird nach Kette2 kopiert.\n");
14    while (Kette1[i] != '\0')
15    {
16       Kette2[i] = Kette1[i];
17       i++;
18    }
19    Kette2[i] = '\0';  /* Abschlusszeichen setzen */
20    printf("Inhalt von Kette2: %s\n", Kette2);
21
22    return 0;
23 }
```

Nun kommen wieder einige Abkürzungen und Kurzformen, die hier Schritt für Schritt erläutert werden. Im ersten Schritt wird die Bedingung der `while`-Schleife verkürzt. Die Bedingung ist ein logischer Ausdruck, der entweder gleich Null (entspricht *falsch*) oder ungleich Null bzw. 1 (entspricht *wahr*) ist. Die Bedingung in `while (Kette[i] != '\0')` fragt nach dem Nullzeichen `'\0'`. Da das Nullzeichen auch den ASCII-Wert 0 hat, kann die Bedingung auch als logische Abfrage gelesen werden: Solange `Kette1[i]` ungleich falsch, dann ... D.h. steht in `Kette1[i]` das Nullzeichen, ist bereits der Ausdruck `Kette1[i]` falsch! Wird noch das `i++;` in die Zuweisung der vorigen Zeile mit hineingeschrieben, sieht die `while`-Schleife folgendermaßen aus (der Rest ändert sich nicht.):

```
while (Kette1[i])
   Kette2[i] = Kette1[i++];
Kette2[i] = '\0'; /* Abschlusszeichen setzen */
```

Insgesamt sind damit bereits zwei Zeilen entfallen. Im zweiten und schon letzten Schritt wird ausgenutzt, dass das Ergebnis der Zuweisung `Kette2[i] = Kette1[i++];` gleich dem Wert von `Kette1[i]` ist (durch das `i++` wird die Variable `i` erst nach der Zuweisung um eins erhöht). Ist also das Ergebnis der Zuweisung gleich Null, ist das Ende der Zeichenkette erreicht. Da die Zuweisung aber vor der Abfrage durchgeführt wird, wird auch das Nullzeichen automatisch in die andere Zeichenkette mit kopiert. Damit entfällt das Setzen des Abschlusszeichens. Damit ist die `while`-Schleife zum Kopieren von Texten ein Zweizeiler geworden (nicht vergessen: die `while`-Schleife benötigt damit eine Leeranweisung (Semikolon)):

```
while (Kette2[i] = Kette1[i++])
   ;
```

So wie das hier gezeigte Kopieren von Texten werden alle Operationen mit Texten programmiert: Eine `while`-Schleife mit der Abbruchbedingung "*solange, bis Nullzeichen (Textende)*" und innerhalb der Schleife jedes einzelne Zeichen bearbeiten. In der Bibliothek `string.h` stehen dem Programmierer bereits eine ganze Reihe von Textoperationen zur Verfügung. Diese Bibliothek wird genauso wie die Standardbibliothek `stdio.h` eingebunden, nämlich durch die Zeile

```
#include <string.h>
```

7.3. Strukturen

Bei Arrays werden viele Elemente desselben Datentyps zusammengefasst. Häufig ist aber gewünscht, auch Daten unterschiedlicher Typen zu einem Objekt zusammenzufassen, wenn diese logisch zusammengehören. Die Lösung dazu hat in C/C++ den Namen *structure*, kurz `struct` und ist folgendermaßen definiert:

```
struct [Typname]
{
   Aufbau
} [Variablenliste];
```

Man kann sich dies gut als Karteikarte vorstellen. Um zum Beispiel eine Bücherverwaltung für die Bibliothek zu erstellen, müssen die Daten eines Buches - Titel, Autor, Standort, usw. - als ein Objekt behandelt werden. Diese einzelnen Daten werden im *Aufbau* wie gewöhnliche Variablen angegeben. Im folgenden Beispiel wird mit Hilfe von `struct` ein Datentyp namens *Buch* (*Typname*) definiert; gleichzeitig wird ein Array dieses Datentyps mit dem Namen *Buecher* erstellt (*Variablenliste*).

Beispiel:

```
struct Buch
{
   char Titel[100];
   char Autor[100];
   char ISBN[20];
   char Standort[10];
   float Preis;
} Buecher[50];      /* 50mal struct Buch */
```

Wahlweise kann der *Typname* oder die *Variablenliste* weggelassen werden. Wird der *Typname* weggelassen, werden Variablen des strukturierten Datentyps definiert, aber dieser strukturierte Datentyp hat keinen Namen. Dadurch können später im Programm keine weiteren Variablen dieses Datentyps definiert werden, es sei denn, es wird wieder die komplette Typdefinition geschrieben.

Wird anders herum die *Variablenliste* weggelassen, wird ein strukturierter Datentyp mit Namen definiert; es werden aber keine Variablen dieses Typs definiert. Dies kann aber später im Programm noch erfolgen, wie das nächste Beispiel unten zeigt.

Auf die Elemente (auch **Felder** oder **Komponenten** genannt) eines strukturierten Datentyps kann nicht direkt zugegriffen werden, weil sie nur in Verbindung mit dem Objekt existieren. Die Anweisung `Preis = 19.99;` ist unsinnig, weil nicht klar ist, welches Buch diesen Preis hat. Der Zugriff auf die Elemente einer Variablen vom Typ `struct` geschieht über einen Punkt zwischen Variablen- und Elementnamen, z.B. `Buch1.Preis = 19.99;`. Dann gilt der Preis für das Buch `Buch1`.

Beispiel:

kap07_07.c

```
01 /*****************************************
02  * Dieses Programm legt zwei Variablen
03  * eines strukturierten Datentyps an.
04  *****************************************/
05 #include <stdio.h>
06 #include <string.h>
07
08 int main()
09 {
10    struct Buch
11    {
12       char Titel[100];
```

```
13        char Autor[100];
14        char Standort[10];
15        float Preis;
16    } Buch1;           /* Variablen-Definition direkt    */
17                       /* bei der Definition des structs */
18    struct Buch Buch2; /* Definition ueber Typnamen      */
19
20    strcpy(Buch1.Titel, "Das indische Tuch");
21    strcpy(Buch1.Autor, "Edgar Wallace");
22    strcpy(Buch1.Standort, "Regal 5");
23    Buch1.Preis = 14.99;
24
25    printf("Das Buch '%s' von '%s' ", Buch1.Titel, Buch1.Autor);
26    printf("kostet %.2f EUR.\n", Buch1.Preis);
27
28    return 0;
29 }
```

7.4. Aufzählungstypen

Häufig gibt es nicht-numerische Wertebereiche. So kann beispielsweise ein Wochentag nur die Werte Sonntag, Montag, ..., Samstag annehmen. Eine mögliche Hilfskonstruktion ist das Verwenden von ganzen Zahlen (`int`), wobei jeweils eine Zahl einem nicht-numerischen Wert entspricht (z.B. 0 = Sonntag, 1 = Montag, usw.). Dabei muss die Bedeutung der einzelnen Zahlen irgendwo als Kommentar festgehalten werden. Dadurch ist das Programm wieder deutlich schlechter lesbar. Auch ist die Zuordnung an einen nicht-numerischen Wert nicht eindeutig. Sind z.B. noch die Monate auf die gleiche Art und Weise in Zahlen "verschlüsselt", ist nicht klar, ob die Zahl 1 nun ein Montag oder der Januar ist. Und schließlich ist es möglich, einer Wochentag-Variablen auch den Wert 27 zuzuweisen, der keinem Wochentag entspricht.

Die Lösung für solche Fälle sind **Aufzählungstypen** (manchmal auch **Enumerationstypen** genannt). Die Syntax dazu sieht so aus:

`enum [`*`Typname`*`] {`*`Aufzählung`*`} [`*`Variablenliste`*`];`

Ähnlich wie bei `struct` kann hier wahlweise der *Typname* oder die *Variablenliste* weggelassen werden. Sinnvoll ist meist nur das Weglassen der *Variablenliste*. Das Weglassen des *Typnamens* ist nur dann sinnvoll, wenn der Aufzählungstyp nur für eine Variablendefinition benötigt wird. Dies wird auch **anonyme Typdefinition** genannt.

Beispiel:

```
enum Wochentag {Sonntag, Montag, Dienstag, Mittwoch,
                Donnerstag, Freitag, Samstag};
```

Damit ist der Aufzählungstyp definiert. Nun lassen sich Variablen dieses Typs definieren:

```
enum Wochentag Werktag, Feiertag, Heute = Dienstag;
```

Diesen Variablen können nur Werte aus der Werteliste des Aufzählungstypen zugewiesen werden. Intern werden sie auf die ganzen Zahlen - beginnend bei 0 - abgebildet. Dadurch ist eine implizite Typumwandlung von einem Aufzählungstypen zu den ganzen Zahlen möglich, aber nicht umgekehrt!

```
Werktag = Montag;           /* richtig */
Feiertag = Sonntag;         /* richtig */
int i = Freitag;            /* richtig (implizite Typumwandlung) */
Heute = 4;                  /* falsch! (keine Typumwandlung möglich!) */
int i = Montag + Dienstag;  /* richtig (aber sinnlos) */
Heute = Montag + Dienstag;  /* falsch! */
Heute++;                    /* falsch! */
```

Die Werte der Werteliste von Aufzählungstypen sind Konstanten und können nach der Definition nicht mehr verändert werden. Wohl aber kann bei der Definition des Aufzählungstypen den Werten andere Zahlen zugewiesen werden. Dazu wird jedem Wert gleich die zugehörige Zahl zugewiesen. Jede Zahl darf dabei natürlich nur einmal verwendet werden.

Beispiel:

```
enum Farben {weiss = 0, blau = 2, gruen = 5, rot = 25,
             gelb = 65, lila = 90, pink = 99};
```

Da die Zahlenwerte der Aufzählungstypen Konstanten sind (d.h. sie haben nur einen Namen und einen Wert, aber keine Speicheradresse), können Aufzählungstypen auch für die Definition von Zahlen-Konstanten verwendet werden.

Beispiel:

```
#define MAX 20
// ist demnach gleichwertig mit
enum { MAX = 20 };
```

7.5. Unions

Unions sind Strukturen, in denen die verschiedenen Elemente *denselben* Speicherplatz bezeichnen, d.h. die Elemente werden alle überlagert. Der benötigte Speicherplatz einer Variablen des Typs `union` ist identisch mit dem Speicherplatzbedarf des größten Elements.

Die Syntax, der Aufbau und der Zugriff auf Elemente des Typs `union` ähneln sehr stark denen von `struct`:

```
union [Typname]
{
   Aufbau
} [Variablenliste];
```

Im folgenden Beispiel werden eine `int`-Zahl und ein `char`-Array überlagert. Da jedes `char` nur ein Byte belegt, kann das Array genauso viele Elemente haben wie die `int`-Zahl Bytes belegt, nämlich `sizeof(int)`.

Beispiel:

kap07_08.c

```
01 #include <stdio.h>
02
03 int main()
04 {
05    int i;
06    union Ueberlagerung
07    {
08       int Zahl;
09       unsigned char c[sizeof(int)];
10    } u;
11
12    do
13    {
14       printf("Zahl eingeben (0 = Ende): ");
15       scanf("%i", &u.Zahl);
16       printf("\nByteweise Darstellung von %i:\n", u.Zahl);
17       for (i = sizeof(int) - 1; i >= 0; i--)
18          printf("Byte Nr: %i = %i\n", i, u.c[i]);
19       printf("\n");
20    } while (u.Zahl);
```

```
21
22      return 0;
23  }
```

Die beiden Ausdrücke `u.Zahl` und `u.c[]` beziehen sich auf denselben Bereich im Speicher, nur die Interpretation ist unterschiedlich. Dieses Beispiel gibt die ganze Zahl Byte für Byte aus, so wie sie im Speicher steht.

7.6. Bitfelder

Gerade in der hardwarenahen und Systemprogrammierung werden häufig Bitfelder benötigt, in denen die einzelnen Bitpositionen unterschiedliche Bedeutungen haben. Ein Bitfeld ist in C/C++ ein `struct`, bei dem bei jedem Element - getrennt mit einem Doppelpunkt - die benötigte Anzahl von Bits dahintergeschrieben wird: *`Datentyp Elementname: Konstante;`*, wobei der *Datentyp* nur einer der Datentypen `char`, `int`, `long` sowie deren `unsigned`-Varianten und ein Aufzählungstyp sein darf.

Beispiel:

```
struct Bitfeld
{
   unsigned int a: 2;
   unsigned int b: 2;
   unsigned int c: 4;
} abc;
```

Auf die Elemente dieser `struct`-Variable kann wie mit jedem anderen `struct` umgegangen werden. Zu beachten sind allerdings die Zahlenbereiche der einzelnen Elemente: `a` und `b` haben jeweils den Zahlenbereich von 0 bis 3 (2-Bit-Zahlen) und das Element `c` den Zahlenbereich von 0 bis 15 (4-Bit-Zahl).

Zusammen sind es 8 Bits. Es dürfen allerdings keine Annahmen gemacht werden, wo diese 8 Bits innerhalb eines 16- bzw. 32-Bit-Wortes angeordnet sind. Dies kann von Compiler zu Compiler verschieden sein. Bitfelder sollten nicht zum Sparen von Arbeitsspeicher verwendet werden, weil der Aufwand des Zugriffs auf einzelne Bits beträchtlich sein kann. Und es nicht einmal sicher, ob wirklich Speicher gespart wird: Es *könnte* sein, dass ein Compiler alle Bitfelder - auch die der Länge 1 - stets an einer 32-Bit-Wortgrenze beginnen lässt. Üblich ist jedoch (ohne Garantie), dass aufeinanderfolgende Bitfelder aneinandergereiht werden.

8. Zeiger

Zeiger gehören eigentlich zu den elementaren Datentypen.

Die bisher vorgestellten Datentypen ermöglichen es, unabhängig von Speicheradressen zu arbeiten. Es wurde nur mit dem Namen der Variablen gearbeitet und nur das Programm selber wusste, welche Speicheradressen sich hinter den Variablennamen verbergen.

Zu einer Variablen gehören folgende Dinge:

- der Name (*kann, muss aber nicht!*),
- die Speicheradresse, an der der Wert der Variablen gespeichert ist,
- der Inhalt der Variable, also der Wert an der Speicheradresse und
- der Datentyp, um den Inhalt zu interpretieren und um die Anzahl der Speicherzellen zu bestimmen.

Beispiel:

Name	Adresse (hex.)	Inhalt	Datentyp	Interpretation
Zahl1	00003D24h	00000000 00001011	`short`	11
	00003D25h			
Zahl2	00003D26h	00000000 00011110	`short`	30
	00003D27h			
Zeichen	00003D28h	01000010	`char`	'B'

Die Adressen in diesem Beispiel sind frei erfunden; jede Ähnlichkeit mit vorhandenen Speicheradressen ist rein zufällig! :-)

Jede Speicheradresse kann 1 Byte = 8 Bits aufnehmen. Die beiden Zahlen im Beispiel sind `short`-Zahlen (16 Bit-Zahlen) und belegen daher 2 Speicheradressen. Dagegen hat die `char`-Variable nur 8 Bits und belegt damit auch nur eine Speicheradresse.

8.1. Zeiger und Adressen

In C/C++ wird an vielen Stellen mit **Zeigern** (im engl. **Pointer**) gearbeitet, um unabhängig von Variablennamen auf die Inhalte der Variablen zugreifen zu können. Ein Zeiger "zeigt" auf eine Variable, d.h. ein Zeiger ist eine Zahlenvariable und beinhaltet als Wert die Adresse der Variablen, auf die sie zeigt. Dadurch benötigen Zeiger immer den gleichen Speicherplatz (z.B. 32 Bit unter einem 32Bit-Betriebssystem).

Bei der Deklaration/Definition von Zeigern wird der Datentyp angegeben, auf den der Zeiger zeigen soll. Vor den Variablennamen von Zeigern wird ein Sternchen (`*`) gesetzt.

Beispiel:

```
int *ip;
```

Diese Zeile bewirkt, dass ein Zeiger mit dem Namen `ip` deklariert und definiert wird. Dieser Zeiger zeigt jetzt grundsätzlich auf Variablen vom Typ `int`. Diese Datentypangabe ist wichtig, da mit ihr nicht nur auf die angegebene Speicheradresse, sondern auch noch auf die nächsten 3 zugegriffen wird (weil eine `int`-Zahl 32 Bits = 4 Bytes = 4 Speicheradressen beinhaltet). Im nächsten Beispiel wird mit einem Zeiger auf eine `int`-Variable gezeigt.

Beispiel:

```
int i;     /* Deklaration der int-Zahl i */
int *ip;   /* Deklaration des Zeigers ip */
```

```
ip = &i;    /* Speicheradresse von i in ip speichern */
i = 99;
*ip = 100; /* weist i die Zahl 100 zu, da ip auf i zeigt */
```

Der unäre **Adressoperator** `&` liefert die Speicheradresse eines Objekts bzw. einer Variablen. Es wird in der Literatur manchmal auch fälschlicherweise von einer Referenz auf das Objekt bzw. vom Referenzieren gesprochen. Unter dem Begriff Referenz wird jedoch in C++ etwas anderes verstanden!

Im Beispiel wird also die Speicheradresse der Variable `i` dem Zeiger `ip` zugewiesen. Danach wird der Variable `i` der Wert 99 zugewiesen. In der letzten Zeile wird dem Inhalt der Variablen, auf die der Zeiger `ip` zeigt, der Wert 100 zugewiesen. Da der Zeiger auf die Variable `i` zeigt, hat `i` anschließend den Wert 100. Dies wird durch den unären **Variablenoperator** `*` erreicht. Es wird auch fälschlicherweise vom Dereferenzieren (siehe oben) gesprochen. Im Speicher sieht es nach dem Ablauf des Programms folgendermaßen aus (die Speicheradressen sind wieder rein zufällig gewählt):

Name	Adresse (hex.)	Inhalt	Datentyp
i	00004711h	100	int
	00004712h		
	00004713h		
	00004714h		
ip	00004715h	00004711h	int*
	00004716h		
	00004717h		
	00004718h		

Hier noch einmal eine Tabelle mit verschiedenen Ausdrücken und die im Beispiel resultierenden Werte:

Ausdruck	Ergebnis
i	100
&i	00004711h
ip	00004711h
*ip	100
&ip	00004715h

Der Variablenoperator liefert aber nicht nur den Zugriff auf den Wert der Variable, auf die er zeigt, sondern er liefert die komplette Variable (daher auch der Begriff Variablenoperator)! D.h. es kann auch wieder auf die Adresse der Variablen zugegriffen werden, wie das folgende Beispiel zeigt.

Beispiel:

kap08_01.c

```
01 #include <stdio.h>
02
03 int main()
04 {
05    int i = 5;
06    int *ip = &i;
07
08    printf("    ip = %p\n", ip);      /* Liefert  */
09    printf("  &*ip = %p\n", &*ip);    /* immer    */
10    printf("  *&ip = %p\n", *&ip);    /* die      */
11    printf("&*&*ip = %p\n", &*&*ip); /* gleiche  */
12    printf("*&*&ip = %p\n", *&*&ip); /* Adresse! */
13
```

```
14     return 0;
15 }
```

Dabei ist zu sehen, dass sich Adressoperator und Variablenoperator aufheben, egal ob erst der Adressoperator und dann der Variablenoperator steht oder umgekehrt; in allen Fällen wird die gleiche Adresse – nämlich die Speicheradresse der Variablen `i` – ausgegeben.

8.2. Der Zeigerwert NULL

Es gibt einen speziellen Zeigerwert, nämlich den Wert `NULL` (definiert in der Headerdatei `stdio.h`). Dieser Zeigerwert zeigt nicht irgendwo hin, sondern definitiv auf "nichts". Die meisten Compiler definieren den `NULL`-Zeiger als einen Zeiger, der auf die Speicheradresse 00000000h zeigt, also auf die allererste Speicheradresse im Arbeitsspeicher. Dieser Zeigerwert kann abgefragt werden, z.B.

```
if (ip == NULL) ....
```

Genauso wie die "normalen" Variablen haben Zeiger nach der Definition einen unbekannten Wert, d.h. sie "zeigen" irgendwo hin. Um dies zu vermeiden, sollten Zeiger immer mit dem Zeigerwert `NULL` initialisiert werden.

8.3. Typprüfung

In C wird – im Gegensatz zu C++ und anderen Programmiersprachen – der Typ eines Zeigers **nicht** geprüft. Es ist also möglich, einen `int`-Zeiger plötzlich auf ein `char` zeigen zu lassen.

Wird mal ein Zeiger benötigt, bei dem noch nicht feststeht, auf welchen Datentypen er zeigen soll, muss ein Zeiger auf `void` verwendet werden.

Beispiel:

```
int *ip = NULL;  /* Zeiger auf int */
char *cp = NULL; /* Zeiger auf char */
void *vp;        /* Zeiger auf void */

vp = ip;  /* korrekt */
vp = cp;  /* korrekt */
/* umgekehrt: auch korrekt (aber nur in C!) */
ip = vp;  /* nur in C korrekt */
cp = vp;  /* nur in C korrekt */
```

In C++ können die umgekehrten Zuweisungen nur mit einer expliziten Typumwandlung durchgeführt werden. Dies sollte aber nicht ohne wichtigen Grund gemacht werden, weil damit die Typkontrolle des C++-Compilers umgangen wird und somit eine weitere Fehlerquelle gegeben ist.

Beispiel:

```
int *ip = NULL;   /* Zeiger auf int */
void *vp = NULL;  /* Zeiger auf void */
ip = (int *) vp;  /* jetzt auch in C++ korrekt! */
```

8.4. Zeiger-Arithmetik

Mit **Zeiger-Arithmetik** wird die Addition und Subtraktion von Zeigern mit ganzen Zahlen bezeichnet. Auch lassen sich zwei Zeiger subtrahieren – vorausgesetzt, sie zeigen auf den gleichen Datentypen.

Beispiel:

```
int i1 = 5, i2 = 7;
int *ip1 = &i1, *ip2 = &i2;
```

```
ip2 = ip1 + 1;  /* erlaubt */
ip1 = ip2 - 1;  /* erlaubt */
ip2 = 1 + ip1;  /* erlaubt */
ip1 = 1 - ip2;  /* Fehler! */
i1 = ip1 + ip2; /* Fehler! */
i2 = ip2 - ip1; /* erlaubt */
```

Alle anderen Operationen wie Multiplizieren usw. dürfen nicht auf Zeiger angewendet werden.

Dabei wird bei der Zeiger-Arithmetik nicht in Bytes sondern in Anzahl von Elementen des Datentyps, auf den der Zeiger zeigt, gerechnet. Wird also zu einem `int`-Zeiger eine 1 addiert, wird nicht auf die nächste Speicheradresse, sondern auf den nächsten `int` gezeigt.

Beispiel:

```
int i = 100;
int *ip1 = &i;
int *ip2 = ip1 + 1;
```

In diesem Beispiel zeigt der Zeiger `ip2` nach der Initialisierung (`ip1 + 1`; also 00004711h + 1) nicht auf die Adresse 00004712h sondern ein Element (also ein `int`) weiter auf die Adresse 00004715h.

Name	Adresse(hex.)	Inhalt	Datentyp
i	00004711h	100	int
	00004712h		
	00004713h		
	00004714h		
ip1	00004715h	00004711h	int*
	00004716h		
	00004717h		
	00004718h		
ip2 =ip1 + 1	00004719h	00004715h	int*
	0000471ah		
	0000471bh		
	0000471ch		

Entsprechend liefert die Differenz von zwei Zeigern nicht die Anzahl der dazwischen liegenden Bytes, sondern die Anzahl der dazwischen liegenden Datenelementen.

Bei einem `void`-Zeiger steht der Datentyp, auf den der Zeiger zeigt, nicht fest. Es wird hier in Bytes gerechnet. Wenn also im obigen Beispiel der Zeiger `ip2` ein Zeiger auf `void` wäre, würde er auf die Adresse 00004712h zeigen.

```
void *ip2 = ip1 + 1; /* zeigt so auf 00004712h! */
```

Hinweis:
Bei der Zeigerarithmetik muss darauf geachtet werden, dass der resultierende Zeiger auf eine Adresse zeigt, die innerhalb des Programmsegments oder innerhalb eines reservierten Speicherbereichs liegt. Ansonsten kommt es schnell zu einem Speicherzugriffsfehler!

8.5. Zeiger und Arrays

Zeiger und Arrays haben vieles gemeinsam. Tatsächlich setzen die meisten Compiler alle Array-Befehle in Zeiger-Befehle um. Der Name eines Arrays (also der Variablenname) ohne eckigen Klammern und Indexangaben gibt die Startadresse des Arrays zurück, d.h. die Adresse des ersten Elements. Er ist damit wie ein Zeiger einsetzbar. Im Gegensatz zu einem richtigen Zeiger kann ihm allerdings keine andere Adresse zugeordnet werden; er ist kein L-Wert (siehe Kapitel *Variablen* Abschnitt *L-Werte und R-Werte*).

Beispiel:

```
int *IntZeiger = NULL; /* Zeiger auf int */
int IntArray[5];       /* Array von int */

IntZeiger = IntArray;  /* Zeiger auf Array-Startadresse */
IntArray[0] = 5;       /* ist identisch mit folgender Zeile */
*IntZeiger = 5;
```

Es kann aber nicht nur auf die Startadresse des Arrays mit Zeigern zugegriffen werden, sondern auch auf jedes andere Element des Arrays. Dazu wird die **Zeiger-Arithmetik** verwendet (siehe vorigen Abschnitt). Zum besseren Verständnis wird das obige Beispiel erweitert.

Beispiel:

```
int *IntZeiger = NULL; /* Zeiger auf int */
int IntArray[5];       /* Array von int */

IntZeiger = IntArray;  /* Zeiger auf Array-Startadresse */
IntArray[0] = 5;       /* ist identisch mit folgender Zeile */
*IntZeiger = 5;
IntArray[1] = 4;       /* ist identisch mit folgender Zeile */
*(IntZeiger + 1) = 4;
```

In der letzen Zeile wird zu dem Wert des Zeigers noch eine 1 dazuaddiert. Damit zeigt dieser Zeiger nicht auf die nächste Speicheradresse, sondern auf das nächste Element im Array (alle Elemente eines Arrays - auch mehrdimensionale - liegen direkt hintereinander im Speicher). Im Beispiel liegt diese Adresse nicht eine sondern 4 Speicheradressen weiter, da der Datentyp `int` einen Speicherbedarf von 4 Byte hat. Das bedeutet, dass bei der Zeiger-Arithmetik der Speicherbedarf des Datentyps, auf den der Zeiger zeigt, bei der Addition und Subtraktion berücksichtigt wird.

Array	Zeiger	Adresse (hex.)	Inhalt
IntArray[0]	*IntZeiger -->	00004711h	5
		00004712h	
		00004713h	
		00004714h	
IntArray[1]	*(IntZeiger + 1) -->	00004715h	4
		00004716h	
		00004717h	
		00004718h	

Mit Hilfe des Inkrement-Operators (++) kann diese letzte Zeile des obigen Beispiels auch folgendermaßen geschrieben werden:

```
*(++IntZeiger) = 4;
```

Allerdings zeigt jetzt der Zeiger nicht mehr auf die Startadresse des Arrays, da das Inkrementieren den Zeiger selbst verändert hat. Im Beispiel zeigt der Zeiger nun nicht auf die Adresse 00004712h sondern auf die Adresse 00004715h.

8.6. Zeiger und Zeichenketten

Da Zeichenketten ein Spezialfall von Arrays sind, gilt der ganze vorige Abschnitt auch für Zeichenketten. Hier sollen noch einige Beispiele zur Verwendung von Zeigern und Zeichenketten zur Verdeutlichung vorgestellt werden.

Beispiel:

```
/* Textlänge ermitteln: */
char Text[] = "Dies ist ein Textarray!";
char *TextZeiger = Text;
int TextLaenge = 0;

while (*TextZeiger++)
   TextLaenge++;
printf("Textlänge: %i",TextLaenge);
```

In der Bedingung der `while`-Schleife wird das erste Zeichen in der Zeichenkette (dahin zeigt der Textzeiger) geprüft. Ist dieses wahr (also ungleich Null und damit ungleich dem Nullzeichen für Textende), wird die Schleife ausgeführt. Zuvor wird noch der Zeiger auf das nächste Zeichen gesetzt (also der Zeiger inkrementiert). In der Schleife wird die Variable `TextLaenge` um eins erhöht. Nun wird in der Bedingung das zweite Zeichen geprüft usw. Ist das Textende erreicht, ist das Zeichen, auf das der Zeiger zeigt, das Nullzeichen, das den ASCII-Wert Null hat. Null bedeutet aber auch falsch, womit die `while`-Schleife nun abgebrochen wird. In der Variablen `TextLaenge` steht damit die Anzahl der Zeichen in der Zeichenkette.

Beispiel:

```
/* Text kopieren: */
char Text1[] = "Dieser Text soll kopiert werden!";
char Text2[35], *TextZeiger1 = Text1, *TextZeiger2 = Text2;

while (*TextZeiger2++ = *TextZeiger1++)
   ;
printf("kopierter Text: %s",Text2);
```

In dieser Schleife wird eine Zeichenkette kopiert. Als Bedingung wird die Zuweisung des ersten Zeichens der Originalzeichenkette `Text1` nach `Text2` durchgeführt. Anschließend werden beide Zeiger ums eins erhöht; sie zeigen damit auf das nächste Zeichen. Das Ergebnis der Zuweisung ist das kopierte Zeichen selber und wird jetzt als Bedingung geprüft. Ist es wahr (also ungleich Null und damit ungleich dem Nullzeichen), wird die Schleife durchgeführt (Leeranweisung). Dann wird das nächste Zeichen kopiert usw. Ist das Textende erreicht, ist das Zeichen, auf das der Zeiger zeigt, das Nullzeichen. Auch dieses wird kopiert. Dann werden die Zeiger ums eins erhöht und das kopierte Zeichen geprüft. Das Nullzeichen hat den ASCII-Wert Null und dadurch ist die Bedingung falsch; die Schleife wird abgebrochen. Das Beispiel kopiert also den vollständigen Text einschließlich des Nullzeichens; die Zeiger zeigen anschließend auf das erste Zeichen nach dem Nullzeichen!

8.7. Zeiger und Strukturen

Mit Zeigern kann auf alle Objekte gezeigt werden, so auch auf Strukturen (`struct`). Dabei verändert sich die Schreibweise, wenn mit Zeigern auf Felder einer Struktur zugegriffen wird. Anstelle des Punktes zwischen Strukturnamen und Feld wird nun ein Pfeil - bestehend aus Minuszeichen und Größer-als-Zeichen (->) - verwendet. Das folgende Beispiel verdeutlicht dieses.

Beispiel:

```
struct Buch
{
   char Titel[100];
   char Autor[100];
   char ISBN[20];
   char Standort[10];
   float Preis;
} Buecher[50]; /* 50mal struct Buch */
struct Buch *BuchZeiger;

BuchZeiger = Buecher;       /* zeigt auf's 1. Element des Arrays */
Buecher[0].Preis = 9.99;    /* ist identisch mit */
```

```
(*BuchZeiger).Preis = 9.99; /* ist identisch mit */
BuchZeiger->Preis = 9.99;
```

Hier gilt es, ganz genau hinzusehen und einen Teilausdruck nach dem anderen auszuwerten, um zu verstehen, was tatsächlich passiert. Ansonsten ändert sich beim Umgang mit den Strukturen gar nichts.

8.8. Unveränderbare Zeiger

Ein Zeiger kann unveränderbar sein (d.h. er kann auf keine andere Adresse zeigen) oder auf eine unveränderbare Variable zeigen – oder beides; je nachdem, an welcher Stelle das Schlüsselwort `const` verwendet wird.

Beispiel:

```
int       i1 = 5;
int const i2 = 3;

// veränderbarer Zeiger auf veränderbare Variable:
int       *       ip1 = &i1;

// veränderbarer Zeiger auf unveränderbare Variable:
int const *       ip2 = &i2;

// unveränderbarer Zeiger auf veränderbare Variable:
int       * const ip3 = &i1;

// unveränderbarer Zeiger auf unveränderbare Variable:
int const * const ip4 = &i2;

*(ip1++); // erlaubt!
(*ip1)++; // erlaubt!
*(ip2++); // erlaubt!
(*ip2)++; // Fehler - unveränderbare Variable!
*(ip3++); // Fehler - unveränderbarer Zeiger!
(*ip3)++; // erlaubt!
*(ip4++); // Fehler - unveränderbarer Zeiger!
(*ip4)++; // Fehler - unveränderbare Variable!
```

Dabei muss eine Variable, auf die ein Zeiger auf unveränderbare Variable (z.B. `int const *`) zeigt, nicht unbedingt unveränderbar sein. Dies mag vielleicht unsinnig erscheinen, kann aber durchaus sinnvoll sein.

Beispiel:

```
int i = 5;             // veränderbare Variable
int *ip1 = &i;         // Zeiger auf veränderbare Variable
int const * ip2 = &i; // Zeiger auf unveränderbare Variable

i = 7;     // ok
*ip1 = 9;  // ok
*ip2 = 11; // Fehler, da ip2 auf unveränderbare Variable zeigt!
```

Obwohl alle drei Zuweisungen auf die gleiche (veränderbare) Speicheradresse zugreifen, hat der Zeiger `ip2` keine Veränderungsberechtigung. Man sagt auch: Der Zeiger `ip1` bietet eine veränderbare Ansicht (im engl: *non-constant view*) und der Zeiger `ip2` eine unveränderbare Ansicht (im engl. *constant view*).

Anders herum funktioniert es übrigens nicht: Ein Zeiger auf eine veränderbare Variable kann nicht auf eine unveränderbare Variable zeigen.

Beispiel:

```
int const i = 5;
int * ip = &i; // Fehler, da i unveränderbar ist!
```

8.9. Zeiger auf Zeiger

Ein Zeiger kann auch auf einen anderen Zeiger zeigen.

Beispiel:

kap08_02.c

```
01 #include <stdio.h>
02
03 int main()
04 {
05    int Wert = 1234;
06    int *Zeiger_auf_Wert = &Wert;
07    int **Zeiger_auf_Zeiger = &Zeiger_auf_Wert;
08
09    printf("  Wert              = %i\n", Wert);
09    printf(" *Zeiger_auf_Wert   = %i\n", *Zeiger_auf_Wert);
09    printf("**Zeiger_auf_Zeiger = %i\n", **Zeiger_auf_Zeiger);
12
13    return 0;
14 }
```

Ein Zeiger auf einen Zeiger wird durch einen doppelten Variablenoperator (`**`) geschaffen. Diesem Zeiger wird die Adresse eines anderen Zeigers zugewiesen, dem wiederum die Adresse einer Variablen zugewiesen wurde. Das angegebene Beispiel gibt folgendes aus:

```
  Wert              = 1234
 *Zeiger_auf_Wert   = 1234
**Zeiger_auf_Zeiger = 1234
```

Dies kann beliebig fortgesetzt werden: Es wäre also möglich, einen Zeiger auf Zeiger auf Zeiger auf Zeiger usw. zu erzeugen, aber es wird keine sinnvolle Anwendung dafür geben.

9. Funktionen

Bisher waren die Programme noch recht klein. Wird die zu lösende Aufgabe komplexer, so sollte das Programm in übersichtliche Teile zerlegt werden. Diese Teile können dann immer noch so komplex sein, dass sie weiter unterteilt werden usw. Der Entwurf dieser Hierarchie nennt sich **Prinzip der schrittweisen Verfeinerung**.

Diese Teilprogramme werden **Funktionen** genannt. Eine Funktion erledigt eine abgeschlossene Teilaufgabe. Weiterhin müssen der Funktion die zu verarbeitenden Werte mitgegeben werden. Dieses geschieht mit sogenannten **Argumenten** bzw. **Parametern**. Die Funktion liefert das Ergebnis an die aufrufende Funktion zurück. Dieses Ergebnis wird auch **Rückgabewert** genannt. Ein Beispiel ist die Sinus-Funktion `y = sin(x)`: Hier wird der Wert `x` als Parameter der Funktion übergeben und das Ergebnis der Sinus-Funktion wird der Variablen `y` zugewiesen.

Eine Funktion muss nur einmal definiert werden. Danach kann sie beliebig oft durch Nennung ihres Namens (dem **Funktionsnamen**) aufgerufen werden. Eine Funktion ist also wiederverwendbar!

9.1. Funktionsprototypen und –definitionen

Bei Funktionen wird zwischen der Deklaration (Bekanntmachung des Namens) und der Definition (Belegung eines Speicherbereichs) unterschieden. Die Deklaration der Funktionen wird im allgemeinen vor dem Hauptprogramm `main()` geschrieben. Diese Deklarationen werden hier **Funktionsprototypen** genannt. Die Syntax eines Funktionsprototyps sieht folgendermaßen aus:

```
Rückgabe-Datentyp Funktionsname(Datentyp [Parametername], ...);
```

Die Parameter werden bei Funktionsprototypen und -definitionen auch **Formalparameter** genannt. Die Parameteranzahl kann bei jeder Funktion unterschiedlich sein. Es können Funktionen mit einer Parameteranzahl zwischen 0 und "beliebig viele" deklariert und definiert werden. Beim Aufruf der Funktion dagegen muss exakt die bei der Deklaration bzw. der Definition vorgegebene Anzahl der Parameter angegeben werden. Der Name des Parameters muss bei der Deklaration nicht angegeben werden, wohl aber bei der Definition.

Durch die Deklaration "weiß" der Compiler, dass irgendwo eine Funktion mit dem angegebenen Funktionsnamen definiert ist, wieviele Parameter sie hat und welchen Datentyp der Rückgabewert hat. Wird diese Funktion im Programm aufgerufen, kann der Compiler jetzt eine Syntaxprüfung machen, ohne dass die Funktion definiert sein muss.

Erst durch die **Funktionsdefinition** kann die Funktion auch angewendet werden, denn erst hier wird der Quellcode angegeben und wird entsprechend Speicherplatz dafür reserviert. Die Syntax einer Funktionsdefinition sieht folgendermaßen aus:

```
Rückgabe-Datentyp Funktionsname(Datentyp Parametername, ...)
{   Anweisungen;
}
```

Der wichtigste Unterschied in der Syntax zwischen Funktionsprototyp und -definition ist, dass beim Prototyp am Ende der Zeile ein Semikolon steht, während bei der Definition kein Semikolon stehen darf. Dafür folgt bei der Definition in den darauffolgenden Zeilen der Quellcode der Funktion, der sogenannte **Funktionskörper**. Der Quellcode der Funktion wird - auch wenn die Funktion nur eine Anweisung beinhaltet - zwischen einem Paar geschweifte Klammern gesetzt (genauso wie bei der `main`-Funktion).

Beispiel:

kap09_01.c

```
01 #include <stdio.h>
02
03 /* Funktionsprototyp: */
04 double Average(double, double, double);
```

```
05
06 /* Hauptprogramm: */
07 int main()
08 {
09    double a = 4.5, b = 3.1415, c = 7.99;
10
11    /* Aufruf der Funktion in der printf-Anweisung: */
12    printf("Durchschnitt: %f\n", Average(a, b, c));
13
14    return 0;
15 }
16
17 /* Funktionsdefinition: */
18 double Average(double Zahl1, double Zahl2, double Zahl3)
19 {
20    return (Zahl1 + Zahl2 + Zahl3) / 3.0;
21 }
```

Aufgerufen wird die Funktion durch den Funktionsnamen gefolgt den Parametern, die in Klammern gesetzt werden. Die Parameter werden in diesem Fall auch **Aktualparameter** genannt. Die Klammern müssen auch dann gesetzt werden, wenn keine Parameter der Funktion übergeben werden. Die Syntax für einen Funktionsaufruf lautet also

`Funktionsname(Variablenname bzw. Konstante, ...);`

Der Rückgabewert der Funktion kann in einer Variablen gespeichert werden oder wie im Beispiel gleich einer anderen Funktion übergeben werden (`printf` ist auch eine Funktion) oder einfach "vergessen" werden, wenn das Funktionsergebnis nicht benötigt wird. Z.B. liefert auch die `printf`-Funktion ein Ergebnis zurück (nämlich die Anzahl der ausgegebenen Zeichen), aber dieses Ergebnis wird nicht weiter benötigt, also wird es auch nicht weiter verarbeitet.

9.2. Gütigkeitsbereiche und Sichtbarkeit

Grundsätzlich sind alle deklarierten Namen von Variablen, Typen, Konstanten, Funktionen, usw. nach der Deklaration nur innerhalb des Blocks gültig, in dem sie deklariert wurden. D.h. z.B. alle Variablen, die innerhalb einer Funktion deklariert bzw. definiert werden, sind nur innerhalb dieser Funktion bekannt und sichtbar; in jeder anderen Funktion sind diese Variablen unbekannt. Diese Variablen werden **lokale** Variablen genannt; sie sind nur in der lokalen Umgebung (innerhalb des Blocks) bekannt. Nach Verlassen des Blocks werden sie wieder vernichtet, d.h. ihr Speicherplatz wird wieder freigegeben.

Variablen, die außerhalb von allen Blöcken deklariert werden, sind innerhalb der ganzen Datei gültig, d.h. innerhalb der `main`- und innerhalb aller anderen Funktionen, die in der gleichen Quellcodedatei definiert sind. Dadurch, dass diese Variablen global gelten, werden sie auch **globale** Variablen genannt.

Grundsätzlich sollte auf globale Variablen nach Möglichkeit verzichtet werden und statt dessen lieber diese Variablen lokal angelegt und bei Bedarf als Parameter in die Funktionen übergeben werden.

Im folgenden Beispielprogramm gibt der Compiler jeweils eine Fehlermeldung für die Zeilen 13 und 24 aus, da in beiden Fällen die Variablen nicht bekannt (deklariert) sind.

Beispiel:

kap09_02.c

```
01 #include <stdio.h>
02
03 int a;           /* globale Variable */
04
05 void Test(void); /* keine Parameter, kein Rückgabewert */
06
07 int main()
```

```
08 {
09    int b;         /* lokal im Hauptprogramm */
10
11    a = 0;         /* erlaubt, da a global */
12    b = 0;         /* erlaubt, da b in diesem Block deklariert */
13    c = 0;         /* FEHLER!, da c nur in Funktion bekannt */
14    Test();
15
16    return 0;
17 }
18
19 void Test(void)
20 {
21    int c;         /* lokal in dieser Funktion */
22
23    a = 0;         /* erlaubt, da a global */
24    b = 0;         /* FEHLER!, da b nur im Hauptprogramm bekannt */
25    c = 0;         /* erlaubt, da c in dieser Funktion deklariert */
26 }
```

Nach diesen Angaben ist es nun auch möglich, mitten im Programm einen Block einzusetzen und in diesem Block lokale Variablen zu deklarieren bzw. zu definieren. Das sieht dann wie folgt aus.

Beispiel:

kap09_03.c

```
01 #include <stdio.h>
02
03 int main()
04 {
05    int a;     /* lokal im Hauptprogramm */
06
07    a = 0;     /* erlaubt, da a im Hauptprogramm deklariert */
08
09    /* Block innerhalb des Hauptprogramms: */
10    {
11       int b; /* lokal in diesem Block */
12
13       a = 3; /* erlaubt */
14       b = 3; /* erlaubt */
15    }
16    b = 0;     /* FEHLER!, da b nur in dem Block bekannt */
17
18    return 0;
19 }
```

Eine Ausnahme bilden lokale Variable, die innerhalb eines Blocks oder einer Funktion als **statische** Variable definiert werden. Dazu wird vor dem Datentyp zusätzlich das Schlüsselwort `static` verwendet. Diese statischen Variablen werden nach Verlassen des Blocks oder der Funktion nicht vernichtet und haben bei erneutem Aufruf des Blocks oder der Funktion noch den alten Wert. Ferner wird eine evtl. Variableninitialisierung nur bei der erstmaligen Definition ausgeführt; bei allen weiteren Aufrufen wird die Initialisierung ignoriert.

Beispiel:

kap09_04.c

```
01 #include <stdio.h>
02
03 void Test(void);
04
```

```
05 int main()
06 {
07    int i;
08
09    for (i = 0; i < 3; i++)
10       Test();
11
12    return 0;
13 }
14
15 void Test(void)
16 {
17    static int Anzahl = 0;
18    /* wird nur beim 1. Aufruf auf Null gesetzt! */
19
20    Anzahl++;
21    printf("Anzahl = %i\n", Anzahl);
22 }
```

Die Ausgabe des Programms ist

```
Ausgabe = 1
Ausgabe = 2
Ausgabe = 3
```

Ohne das Schlüsselwort `static` würde dreimal eine 1 ausgegeben werden, da die lokale Variable bei jedem Funktionsaufruf wieder neu erzeugt würde.

Die Parameter einer Funktion werden innerhalb der Funktion als lokale Variablen behandelt; von außen betrachtet stellen sie die Daten-Schnittstelle zur Funktion dar. Mehr dazu im nächsten Abschnitt.

9.3. Funktionsschnittstelle

Der Datentransfer in Funktionen hinein (Parameter) und aus Funktionen heraus (Rückgabewert) wird durch die Beschreibung der **Schnittstelle** festgelegt. Unter einer Schnittstelle ist eine formale Vereinbarung zwischen Aufrufer und Funktion über die Art und Weise des Datentransports zu verstehen. Auch das, was die Funktion leistet, gehört zur Schnittstelle (sollte als Kommentar beim Funktionskopf stehen). In diesem Abschnitt soll es nur um den Datentransfer gehen. Die Schnittstelle ist durch den Funktionsprototyp eindeutig beschrieben und enthält folgendes:

- den Rückgabetyp der Funktion,
- den Funktionsnamen,
- Parameter, die der Funktion bekannt gemacht werden inkl. deren Datentypen und
- die Art der Parameterübergabe.

Der Compiler prüft, ob die Definition der Schnittstelle bei einem Funktionsaufruf eingehalten wird.

Für den Datentransfer in die Funktion hinein gibt es zwei verschiedene Arten des Datentransports: Die Übergabe per Wert und die Übergabe per Zeiger.

Übergabe per Wert

Bei der Übergabe per Wert wird der Wert in den sogenannten **Stack** (einem Zwischenspeicher u.a. für die Parameterübergabe) kopiert – daher wird dies manchmal auch Übergabe per Kopie genannt. Innerhalb der Funktion werden die Werte auf dem Stack als lokalen Variable gearbeitet, die am Ende der Funktion wieder vernichtet werden, während die Originale unverändert bleiben.

Beispiel:

kap09_05.c

```c
#include <stdio.h>

int Addiere_3(int x);

int main()
{
   int Ergebnis, Zahl = 2;

   printf("Wert von Zahl = %i\n", Zahl);
   Ergebnis = Addiere_3(Zahl);
   printf("Ergebnis 'Addiere_3(Zahl)' = %i\n", Ergebnis);
   printf("Wert von Zahl = %i (unveraendert)\n", Zahl);

   return 0;
}

int Addiere_3(int x)
{
   x += 3;
   return x;
}
```

Die Übergabe per Wert sollte generell verwendet werden, wenn ein Objekt von der Funktion nicht verändert werden soll. Wenn der Kopiervorgang in den Stack allerdings zu lange dauert (bei sehr großen Objekten oder bei häufigem Aufruf der Funktion), kann auch eine Übergabe per Zeiger geschehen. Dann sollte allerdings sichergestellt werden, dass die Funktion den übergebenen Wert nicht verändert, z.B. durch Verwendung von `const`-Parametern.

Übergabe per Zeiger

Die Übergabe per Zeiger ist ein Spezialfall der Übergabe per Wert, es wird nämlich der Zeiger auf das Objekt im Stack abgelegt. Innerhalb der Funktion wird dieser Zeiger wieder als lokale Variable behandelt. Der Zeiger wird also am Ende der Funktion wieder vernichtet. Aber über diesen Zeiger kann auf das eigentliche Objekt zugegriffen (und damit auch verändert) werden. Es wird also nicht das Objekt selber übergeben, sondern ein Zeiger auf das Objekt.

Bei den Parametern wird vor dem Parameternamen ein Stern gesetzt, da ja ein Zeiger übergeben wird; der Datentyp dagegen wird nicht geändert. Beim Aufruf selber wird vor dem Parameternamen der Adressoperator ('`&`') gesetzt. Dadurch wird ein Zeiger auf das eigentliche Objekt erzeugt und der Funktion übergeben. Das obige Beispiel wird nun so geändert, dass die Zahl per Zeiger an die Funktion übergeben wird.

Beispiel:

kap09_06.c

```c
#include <stdio.h>

int Addiere_3(int *);

int main()
{
   int Ergebnis, Zahl = 2;

   printf("Wert von Zahl = %i\n", Zahl);
   Ergebnis = Addiere_3(&Zahl);
   printf("Ergebnis 'Addiere_3(&Zahl)' = %i\n", Ergebnis);
   printf("Wert von Zahl = %i (veraendert!!!)\n", Zahl);

   return 0;
```

```
15 }
16
17 int Addiere_3(int *x)
18 {
19    *x += 3;
20    return *x;
21 }
```

Rückgabewerte

Der Rückgabewert wird innerhalb der Funktion mit dem Befehl `return Wert;` angegeben, wobei *Wert* der Rückgabewert ist. Gleichzeitig beendet dieser Befehl die Funktion, unabhängig davon, ob weitere Anweisungen folgen oder nicht. Als Beispiel wird noch einmal das obige Beispiel verwendet.

Beispiel:

kap09_07.c

```
01 #include <stdio.h>
02
03 int Addiere_3(int);
04
05 int main()
06 {
07    int Ergebnis, Zahl = 2;
08
09    printf("Wert von Zahl = %i\n", Zahl);
10    Ergebnis = Addiere_3(Zahl);
11    printf("Ergebnis 'Addiere_3(Zahl)' = %i\n", Ergebnis);
12    printf("Wert von Zahl = %i (unveraendert)\n", Zahl);
13
14    return 0;
15 }
16
17 int Addiere_3(int x)
18 {
19    x += 3;
20    return x;
21    printf("STOP!"); /* Diese Anweisung wird nie ausgefuehrt! */
22 }
```

Bei der Rückgabe von Zeigern muss darauf geachtet werden, dass das dazugehörige Objekt auch in der aufrufenden Funktion existiert. Wenn z.B. ein Zeiger auf eine lokale Variable zurückgegeben wird, greift die aufrufende Funktion auf einen Speicherbereich zu, der eventuell bereits von einem anderen Programm oder einer anderen Funktion verwendet wird. Das Ergebnis ist dann u.U. falsch. Im schlimmsten Fall kann sogar ein Systemabsturz erzeugt werden! Das folgende Beispiel zeigt, wie es **falsch** ist, obwohl der Compiler u.U. keine Fehler und keine Warnungen anzeigt und das Programm sogar läuft.

Beispiel:

kap09_08.c

```
01 #include <stdio.h>
02
03 int *Maximum(int, int);
04
05 int main()
06 {
07    int x = 17, y = 4, *zp1, *zp2, z;
08
09    zp1 = Maximum(x, y);
10    zp2 = Maximum(y, x);
```

```
11    z = *zp1;          /* Ergebnis 1. Fkt.aufruf zwischenspeichern */
12    printf("x = %i\n", x);
13    printf("y = %i\n", y);
14    printf("z = %i\n", z);              /* Ergebnis 1. Fkt.aufruf */
15    printf("Maximum(%i, %i) = %i\n", x, y, *zp1); /* 1. Fkt.aufruf */
16    printf("Maximum(%i, %i) = %i\n", y, x, *zp2); /* 2. Fkt.aufruf */
17
18    return 0;
19 }
20
21 int *Maximum(int a, int b)
22 {
23    /* a und b sind lokale Kopien!!! */
24    return a > b ? &a : &b; /* Fehler!!! */
25 }
```

Zuerst wird der Zeiger `zp1` auf die lokale Variable `a` gesetzt (da `x` > `y` ist). Anschließend wird die gleiche Funktion noch einmal aufgerufen. Dabei werden die lokalen Variablen `a` und `b`, die mit hoher Wahrscheinlichkeit die gleichen Speicheradressen belegen, auf andere Werte gesetzt. Damit zeigt auch `zp1` auf einen anderen Wert. Damit ist das Ergebnis in `z` und in `*zp1` falsch!

9.4. Rekursiver Funktionsaufruf

Der Aufruf einer Funktion durch sich selbst wird **Rekursion** genannt. Eine Rekursion muss irgendwann auf eine Abbruchbedingung stoßen, damit die Rekursion nicht unendlich ist. Bei einer unendlichen Rekursion wird irgendwann ein sogenannter **Stacküberlauf** (**stack overflow**) erzeugt; das Programm wird damit abgebrochen.

Als Beispiel für die Verwendung einer Rekursion wird die Quersumme einer ganzen Zahl berechnet: Eine Funktion ermittelt die letzte Ziffer einer Zahl, addiert diese zur Quersumme und ruft sich selbst mit den restlichen Ziffern wieder auf. Das Prinzip dieser Funktion lässt sich in zwei Sätze zusammenfassen:

1. Die Quersumme der Zahl 0 ist gleich 0. Dies ist die Abbruchbedingung für die Rekursion!
2. Die Quersumme einer Zahl ist gleich der letzten Ziffer plus der Quersumme der Zahl, die um diese Ziffer gekürzt wurde.

Die Quersumme von 873956 ist also gleich 6 plus der Quersumme von 87395 und ist damit gleich 6 plus 5 plus der Quersumme von 8739 usw. Auf jede Quersumme wird der Satz 2 angewandt, bis der Satz 1 gilt. Satz 1 gilt dann, wenn alle Ziffern von der Zahl abgetrennt wurden. Die letzte Ziffer der Zahl wird durch *modulo 10* (Divisionsrest) abgetrennt. Die Zahl ohne der letzten Ziffer wird durch eine ganzzahlige Division durch 10 erhalten.

Beispiel:

kap09_09.c

```
01 #include <stdio.h>
02
03 int Quersumme(unsigned long);
04
05 int main()
06 {
07    unsigned long Zahl;
08
09    printf("Bitte eine positive ganze Zahl eingeben: ");
10    scanf("%lu", &Zahl);
11    printf("Quersumme von %u ist %i\n", Zahl, Quersumme(Zahl));
12
13    return 0;
```

```
14 }
15
16 int Quersumme(unsigned long x)
17 {
18    int letzteZiffer = 0;
19
20    if (!x) /* if (x == 0) Abbruchbedingung */
21       return 0;   /* Abbruch der Rekursion */
22    else
23    {
24       letzteZiffer = x % 10;  /* modulo 10 */
25       return letzteZiffer + Quersumme(x / 10); /* Rekursion */
26    }
27 }
```

Die Funktion `Quersumme` kann auch nicht-rekursiv geschrieben werden, in dem eine Schleife verwendet wird. Diese Variante wird **iterativ** genannt.

kap09_10.c

```
01 #include <stdio.h>
02
03 int Quersumme(unsigned long);
04
05 int main()
06 {
07    unsigned long Zahl;
08
09    printf("Bitte eine positive ganze Zahl eingeben: ");
10    scanf("%lu", &Zahl);
11    printf("Quersumme von %u ist %i\n", Zahl, Quersumme(Zahl));
12
13    return 0;
14 }
15
16 int Quersumme(unsigned long x)
17 {
18    int QSumme = 0;
19
20    while (x > 0)
21    {
22       QSumme += x % 10; /* letzte Ziffer addieren  */
23       x /= 10;          /* letzte Ziffer abtrennen */
24    }
25    return QSumme;       /* Rueckgabe der Quersumme */
26 }
```

9.5. *Zeiger auf Funktionen*

Steht zum Zeitpunkt der Compilierung noch nicht fest, welche Funktion zur Laufzeit aufgerufen werden soll (z.B. wenn der Anwender zur Laufzeit erst die gewünschte Funktion auswählt) oder an welcher Adresse die Funktion steht (z.B. wenn Funktionen zur Laufzeit nachgeladen werden und daher deren Adressen beim Kompilieren noch nicht bekannt sind), wird mit Zeigern auf Funktionen gearbeitet.

Das folgende Beispiel soll den Umgang mit Zeigern auf Funktionen verdeutlichen:

Beispiel:

kap09_11.c

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

void Funktionswerte(double (*)(double));

int main()
{
   double (*TrigFkt)(double); /* Zeiger auf Funktion, die ein
                                 double als Parameter erhaelt
                                 und ein double zurueckgibt.      */

   TrigFkt = sin;             /* Zeiger auf sin-Funktion zuweisen */
   Funktionswerte(TrigFkt);

   return 0;
}

void Funktionswerte(double (*Fkt)(double))
{
   static const double PI = 4 * atan(1);
   double x;

   for (x = 0; x < PI; x += 0.01)
      printf("f(%5.2f) = %5.2f\n", x, Fkt(x));
}
```

Auch ein Array von Zeigern auf Funktionen ist möglich. Dazu wird gleich hinter dem Zeigernamen der Index angegeben.

Beispiel:

kap09_12.c

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

void Funktionswerte(double (*)(double));

int main()
{
   double (*TrigFkt[2])(double); /* Array von Zeigern auf Funktionen,
                                    die ein double als Parameter
                                    erhalten und ein double
                                    zurueckgeben.                  */

   TrigFkt[0] = sin;             /* Zeiger auf sin-Fkt. zuweisen   */
   TrigFkt[1] = cos;             /* Zeiger auf cos-Fkt. zuweisen   */
   Funktionswerte(TrigFkt[0]);
   Funktionswerte(TrigFkt[1]);

   return 0;
}

void Funktionswerte(double (*Fkt)(double))
{
   static const double PI = 4 * atan(1);
   double x;

   for (x = 0; x < PI; x += 0.01)
```

```
27        printf("f(%5.2f) = %5.2f\n", x, Fkt(x));
28 }
```

Zeiger auf Funktionen scheinen aber nur unzureichend standardisiert zu sein, da einige Compiler diese Beispiele nicht korrekt verarbeiten können.

9.6. Die Funktion main()

Die Funktion `main()` - das Hauptprogramm - ist eine spezielle Funktion. Jedes C-/C++-Programm startet definitionsgemäß mit `main()`, so dass `main()` in jedem Programm genau einmal vorhanden sein muss. Der Rückgabewert ist standardmäßig `int`, aber viele Compiler lassen auch `void` zu (dann ist natürlich auch kein `return 0;` mehr erlaubt!). Die `main()`-Funktion kann nicht überladen werden. Die zwei folgenden Formen werden von allen Compilern unterstützt.

einfache Form:

```
int main()
{   ...
    return 0; /* Exit-Code */
}
```

allgemeine (komplexere) Form:

```
int main(int argc, char *argv[])
{   ...
    return 0; /* Exit-Code */
}
```

In der zweiten Form werden zwei Parameter übergeben. Über diese beiden Parameter kann auf alle **Kommandozeilenparameter** zugegriffen werden, die beim Aufruf des Programms angegeben wurden. Das folgende Beispiel startet das Programm `prog` mit 4 Kommandozeilenparameter.

Beispiel:

```
./prog 1 Test 547.32 3
```

Der erste Parameter `argc` ist die Anzahl der Kommandozeilenparameter einschließlich des Programmaufrufs, im Beispiel gleich 5. Der zweite Parameter ist ein String-Array, also ein Array von Zeichenketten. In diesen Zeichenketten stehen die einzelnen Kommandozeilenparameter. Für das obige Beispiel sind folgende Werte in diesem Array gespeichert:

```
argv[0] = "./prog"
argv[1] = "1"
argv[2] = "Test"
argv[3] = "547.32"
argv[4] = "3"
argv[5] = NULL
```

Wichtig: Auch die Zahlen-Kommandozeilenparameter werden hier als Texte gespeichert.

Nun folgt ein Beispielprogramm, dass die Anzahl sowie alle Kommandozeilenparameter auflistet.

Beispiel:

kap09_13.c

```
01 #include <stdio.h>
02
03 int main(int argc, char *argv[])
04 {
05    int i = 1;
06
07    printf("Programmaufruf: %s\n", argv[0]);
08    printf("Anzahl Kommandozeilenparameter: %i\n", argc - 1);
```

```
09    printf("Kommandozeilenparameter:\n");
10    while (argv[i])
11    {
12       printf("%2i: %s\n", i, argv[i]);
13       i++;
14    }
15
16    return 0;
17 }
```

10. Präprozessor-Befehle

Präprozessor-Befehle (auch **Compilerdirektiven** genannt) sind Anweisungen an den Präprozessor des Compilers. Der Präprozessor ist dem Compiler vorgeschaltet, d.h. bei jeder Compilierung wird erst der Präprozessor gestartet, der an ihn gerichtete Befehle im Quellcode sucht und verarbeitet. Damit steuert der Präprozessor den Compilierungsvorgang.

Präprozessor-Befehle beginnen stets mit # am Zeilenanfang. Im Gegensatz zu Anweisungen in C/C++ enden sie aber **nicht** mit einem Semikolon! In der folgenden Tabelle sind alle Präprozessor-Befehle aufgelistet:

Präprozessor-Befehl	Bedeutung
`#include`	Fügt Text aus einer anderen Quelltextdatei ein.
`#define`	Definiert ein Makro.
`#undef`	Entfernt ein Makro.
`#if`	Bedingte Compilierung in Abhängigkeit der nachstehenden Bedingung
`#ifdef`	Bedingte Compilierung in Abhängigkeit, ob ein Makroname definiert ist.
`#ifndef`	Bedingte Compilierung in Abhängigkeit, ob ein Makroname nicht definiert ist.
`#else`	Alternative Compilierung, wenn die Bedingung des vorstehenden `#if`, `#elif`, `#ifdef` oder `#ifndef` nicht erfüllt ist
`#elif` [2]	Alternative Compilierung in Abhängigkeit der nachstehenden Bedingung, wenn die Bedingung des vorstehenden `#if`, `#elif`, `#ifdef` oder `#ifndef` nicht erfüllt ist (also eine Art Kombination aus `#else` und `#if`)
`#endif`	Beendet den Block mit der bedingten Compilierung.
`#line`	Setzt die Zeilennummer für Compiler-Meldungen.
`defined` [1]	Liefert eine 1, wenn der nachstehende Makroname definiert ist, sonst eine 0. Kann nur zusammen mit `#if` und `#elif` verwendet werden.
#-O*perator* [1]	Ersetzt innerhalb eines Makros einen Makroparameter durch eine konstante Zeichenkette, die den Wert des Parameters enthält.
##-O*perator* [1]	Erzeugt ein einzelnes Token aus zwei hintereinander stehenden Tokens.
`#pragma` [1]	Gibt Compiler- und System-abhängige Infomationen an den Compiler.
`#error` [1]	Erzeugt einen Fehler während der Compilierung mit der angegebenen Fehlermeldung.

[1] Neu in C99.
[2] Gehört nicht zum Standard-C, obwohl es von vielen Compilern unterstützt wird.

Der Präprozessor entfernt üblicherweise alle Zeilen mit Präprozessor-Befehlen aus dem Quelltext und führt die entsprechenden Befehle aus. Der daraus entstehende Quelltext muss ein gültiges C-Programm ergeben.

Eine Zeile, in der (außer weißen Leerzeichen) nur ein # steht, wird wie eine Leerzeile behandelt. Ältere C-Compiler können hier Fehlermeldungen liefern.

Im Standard-C werden Leerzeichen, die vor dem # sowie zwischen # und dem Präprozessor-Befehl stehen, ignoriert. Ältere C-Compiler ignorieren diese Leerzeichen unter Umständen nicht. Daher sollte an diesen Stellen auf Leerzeichen verzichtet werden.

Alle Präprozessor-Zeilen werden vor der Textersetzung der Makros erkannt. Daher kann ein Makro nicht einen anderen Präprozessor-Befehl erzeugen. Das folgende Beispiel erzeugt eine Fehlermeldung beim Compilieren.

Beispiel:

```
#define GETMATH #include <math.h>
GETMATH
```

Der Präprozessor ersetzt (erst nachdem er alle Präprozessorbefehle erkannt hat) das `GETMATH` durch

`#include <math.h>.`

Diese Zeile bleibt dann für die Compilierung so stehen und erzeugt während der Compilierung einen Fehler.

10.1. #include

Bereits bekannt ist die `#include`-Anweisung. Mit ihr wird an der aktuellen Stelle eine externe Datei eingelesen und in den Quellcode eingefügt. Im allgemeinen werden mit diesem Befehl die Headerdateien (Dateiendung `*.h`) eingefügt, aber auch andere Quellcodedateien (Dateiendung `*.c` oder `*.cpp`) sind möglich. Die gewünschte Datei wird in spitze Klammern (`<...>`) gesetzt. Die angegebene Datei wird im Standardverzeichnis der Headerdateien gesucht. Dieses Verzeichnis wird häufig auch **Includeverzeichnis** genannt. Um auch eigene Headerdateien einlesen zu können, die meistens bei den Quellcodedateien gespeichert werden, wird der Dateiname in Anführungszeichen (`"..."`) anstatt in spitzen Klammern gesetzt. Dabei kann auch das Laufwerk und das Verzeichnis angegeben werden, wenn es nicht bei den Quellcodedateien, sondern gesondert gespeichert wird.

Beispiele:

```
#include <stdio.h>   /* sucht Datei im Includeverzeichnis */
#include "projekt.h" /* sucht Datei im akt. Verzeichnis   */
#define Makroname "my_header.h"
#include Makroname   /* sucht Datei, die durch den
                        Makronamen angegeben ist          */
```

Bei Pfadangaben sollte aus Kompatibilitätsgründen ein Schrägstrich (`/`) anstelle des sonst üblichen Backslashs (`\`) verwendet werden. Natürlich kann auch ein Backslash verwendet werden, dann ist aber das Programm nicht mehr unter UNIX compilierfähig! Ferner müssen bei der Verwendung von Backslashs immer zwei hintereinander geschrieben werden, da der Backslash ein Steuerzeichen (wie z.B. `\n`) einleitet.

Beispiele:

```
#include "C:\\Verz\\header.h"           /* läuft nicht unter UNIX! */
#include "U:/projekt/include/projekt.h"
```

10.2. #define, #ifdef und #ifndef

Für die Fehlersuche werden meist Zeilen in den Quellcode eingefügt, die z.B. den Inhalt von Variablen anzeigen. Ist die Fehlersuche abgeschlossen, müssen alle diese Zeilen wieder gelöscht werden (und bei neuen Fehlern wieder geschrieben werden usw.). Um dieses zu vermeiden, kann für den Präprozessor ein Label definiert werden. Dazu wird der Befehl `#define` *`Label`* verwendet. Dieses Label kann der Präprozessor abfragen, ob es definiert ist oder nicht. Je nachdem kann der Präprozessor für das Compilieren Quellcodebereiche einblenden oder nicht. Diese Abfrage wird mit dem Befehl `#ifdef` *`Label`* (Abkürzung für `#if defined`) durchgeführt. Ist das Label definiert, werden alle folgenden Anweisungen bis zum `#else` oder `#endif` mit compiliert, ansonsten nicht. Um nun die Fehlersuche abzuschließen, braucht nur die Definition der Marke gelöscht oder auskommentiert zu werden. Wird sie nur auskommentiert, kann sie jederzeit mit minimalem Aufwand wieder aktiviert werden.

Genauso wie in C/C++ kann auch ein `else`-Zweig eingerichtet werden. Dazu wird der Befehl `#else` benutzt.

Beispiel:

kap10_01.c

```
01 #include <stdio.h>
02
03 #define TEST
04
05 int main()
06 {
07    int a = 3, b = 5, c;
08
09    c = a * b;
10    #ifdef TEST
```

```
11        printf("Variableninhalte:\n");
12        printf("a = %i\nb = %i\nc = %i\n", a, b, c);
13     #else
14        printf("Ergebnis von %i mal %i ist %i\n", a, b, c);
15     #endif
16
17     return 0;
18  }
```

In diesem Beispiel wird am Anfang des Programms das Label `TEST` (der Name des Labels ist beliebig, unterliegt aber den Regels eines Bezeichners; wird standardmäßig komplett in Großbuchstaben geschrieben) definiert. Im Verlauf des Programms wird dieses Label abgefragt, um den Inhalt der Variablen anzeigen zu lassen. Ist dagegen das Label nicht definiert, wird das Ergebnis der Berechnung angezeigt.

Entsprechend umgekehrt wird mit dem Befehl `#ifndef` *`Label`* (Abkürzung für `#if not defined`) geprüft, ob das Label (noch) nicht definiert ist.

Ein anderer Anwendungsbereich dieser Präprozessor-Befehle ist der **Includewächter** im Bereich der Header-Dateien. Wird in einem großen Projekt dieselbe Header-Datei mehrfach eingelesen, werden die darin enthaltenen Deklarationen bzw. Definitionen mehrfach ausgeführt. Dies führt zu Fehlern bei der Compilierung. Um dies zu verhindern, wird in der Header-Datei zuerst abgefragt, ob ein bestimmtes Label bereits definiert ist. Ist dies nicht der Fall (nämlich beim ersten Durchlauf), wird dieses Label definiert. Dann folgen alle Deklarationen und Definitionen der Header-Datei und am Ende das `#endif`. Wird diese Header-Datei im gleichen Compiliervorgang ein weiteres Mal eingelesen, ist das Label bereits definiert und der Rest der Header-Datei wird nicht wiederholt compiliert.

Das nächste Beispiel zeigt den typischen Aufbau von Header-Dateien. Der Name des Labels, das definiert wird, ist meistens der Dateiname der Header-Datei, wobei der Punkt durch einen Unterstrich ersetzt wird. Dadurch können in mehreren Header-Dateien keine Marken mit dem gleichen Namen existieren. Da mit dem `#define`-Befehl auch Textersetzung gemacht wird (siehe nächster Abschnitt), würde im folgenden Beispiel jedes Vorkommen des Labels - z.B. als Variable - durch "nichts" ersetzt werden. Um dies zu verhindern, wird der Name des Labels gleichzeitig durch sich selbst ersetzt.

Beispiel:

```
/* Headerdatei PROJEKT.H */

#ifndef PROJEKT_H
   #define PROJEKT_H PROJEKT_H

   /* Deklarationen und Definitionen */

#endif /* PROJEKT_H */
```

10.3. Makros mit #define

Ein weiterer Einsatzbereich des Befehls `#define` ist die Textersetzung im Quellcode, wobei auch Parameter erlaubt sind. Diese Textersetzungen werden auch **Makros** genannt. Der Präprozessor ersetzt vor dem Compilierungsvorgang im Quellcode alle Texte, die mit dem angegebenen Text identisch sind, durch den angegebenen Ersatztext. Im folgenden Beispiel wird angegeben, dass im Quellcode alle Vorkommen von `PI` durch die Zahl 3.14159265 ersetzen sollen. Es hat sich eingebürgert, diese Makronamen komplett in Großbuchstaben zu schreiben.

Beispiel:

kap10_02.c

```
01 #include <stdio.h>
02
03 #define PI 3.14159265
```

```
04
05 int main()
06 {
07    printf("PI = %f\n", PI);
08
09    return 0;
10 }
```

Nach der Definition kann `PI` wie eine konstante Variable verwendet werden. Der Compiler bekommt diese aber gar nicht zu sehen, da vor dem Compilierungsvorgang ja alle Stellen, an denen `PI` steht, durch die angegebene Zahl ersetzt. Daher kann übrigens auch kein Zeiger auf `PI` gerichtet werden.

Wie oben erwähnt, können auch Makros mit Parametern eingerichtet werden.

Beispiel:

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

Steht nun im Quelltext die Zeile

```
z = QUADRAT(5);
```

wird diese durch den Präprozessor umgeschrieben in

```
z = 5 * 5;
```

Diese Makros können das Leben eines Programmierers manchmal sehr vereinfachen, aber sie können auch gefährlich sein. Wie gefährlich das angegebene Makro sein kann, ist sehr schnell gezeigt; es braucht nur eine Summe als Parameter übergeben werden. Dann wird aus der Zeile

```
z = QUADRAT(3 + 6);
```

durch den Präprozessor folgende Zeile gemacht.

```
Z = 3 + 6 * 3 + 6;
```

Das Ergebnis ist 27 statt 81! Was fehlt, sind Klammern. Aber auch wenn Klammern bei der Definition des Makros gesetzt werden, kann es zu Problemen kommen.

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

Vorsichtshalber sind nicht nur um die beiden x Klammern, sondern auch um den gesamten Ausdruck gesetzt worden. In der folgenden Zeile wird der Parameter vor der Übergabe um 1 erhöht.

```
z = QUADRAT(++x);
```

Falsch, denn nachdem der Präprozessor die Textersetzung durchgeführt hat, steht folgende Zeile da.

```
z = ((++x) * (++x));
```

Hier wird deutlich, dass der Parameter vor der Multiplikation zweimal um 1 erhöht wird.

Ferner wird mit Makros die Typkontrolle umgangen. Werden nun Parameter mit falschen Datentypen übergeben, wird die Fehlersuche stark erschwert.

Aus `z = QUADRAT("Test");` wird `z = (("Test") * ("Test"));`.

Fazit:

Die Textersetzung mit dem `#define`-Befehl sollte im allgemeinen **nicht** verwendet werden, wenn es Alternativen gibt. Und die gibt es fast immer!

10.4. Variable Argumentenliste in Makrodefinitionen

Mit dem C99-Standard können Funktions-ähnliche Makros mit variabler Parameteranzahl erzeugt werden. Dazu muss als letzter bzw. als einziger Parameter eine Ellipse (`...`) angegeben werden. Diese Parameter können dann im Makro mit dem Bezeichner `__VA_ARGS__` (Kurzfassung von *variable arguments*) verwendet werden.

Im folgenden Beispiel wird der Inhalt der Variablen `x` im Debug-Modus auf dem Bildschirm angezeigt, ansonsten auf dem Fehlerkanal `stderr` ausgegeben.

Beispiel:

kap10_03.c

```
01 #include <stdio.h>
02
03 /* in folgender Zeile "//" entfernen für Debug-Modus */
04 // #define DEBUG
05
06 #ifdef DEBUG
07    #define PRINTF(...) printf(__VA_ARGS__)
08 #else
09    #define PRINTF(...) fprintf(stderr, __VA_ARGS__)
10 #endif
11
12 int main()
13 {
14    int x = 0;
15
16    PRINTF("Variable x = %d\n", x);
17
18    return 0;
19 }
```

Im folgenden Beispiel wird ein Makro mit variabler Parameteranzahl definiert, das die Parameter mit Hilfe des #-Operators (siehe unten im Abschnitt *#-Operator*) in eine Zeichenkette umwandelt. Es werden also die Namen der Parameter und nicht deren Inhalte ausgegeben.

Beispiel:

kap10_04.c

```
01 #include <stdio.h>
02
03 #define MAKE_STRING(...) #__VA_ARGS__
04
05 int main()
06 {
07    int a = 1, b = 2, c = 3, d = 4;
08
09    printf("Die Variablen %s haben ", MAKE_STRING(a, b, c, d));
10    printf("die Werte %i, %i, %i und %i.\n", a, b, c, d);
11
12    return 0;
13 }
```

10.5. Vordefinierte Makros

Im Standard-C müssen bereits einige Makros im Präprozessor vordefiniert sein. Die Namen der vordefinierten Makros beginnen und enden jeweils mit zwei Unterstrichen. Die wichtigsten vordefinierten Makros sind in der folgenden Tabelle aufgelistet.

Vordefiniertes Makro	Bedeutung
`__LINE__`	Zeilennummer innerhalb der aktuellen Quellcodedatei
`__FILE__`	Name der aktuellen Quellcodedatei
`__DATE__`	Datum, wann das Programm compiliert wurde (als Zeichenkette)
`__TIME__`	Uhrzeit, wann das Programm compiliert wurde (als Zeichenkette)

`__STDC__`	Liefert eine 1, wenn sich der Compiler nach dem Standard-C richtet.
`__STDC_VERSION__`	Vor dem C95-Standard war dieses Makro nicht definiert; ab dem C95-Standard liefert dieses Makro das Veröffentlichungsdatum in von Form `yyyymm` als `long`-Zahl: = `199409L` für C95-Standard = `199901L` für C99-Standard = `201112L` für C11-Standard = `201710L` für C17-Standard

Beispiele:

kap10_05.c

```
01 #include <stdio.h>
02
03 int main()
04 {
05    printf("Programm wurde compiliert am ");
06    printf("%s um %s.\n", __DATE__, __TIME__);
07
08    printf("Diese Programmzeile steht in Zeile ");
09    printf("%d in der Datei %s.\n", __LINE__, __FILE__);
10
11    #ifdef __STDC__
12    printf("Standard-C-Compiler!\n");
13    #else
14    printf("Kein Standard-C-Compiler!\n");
15    #endif
16
17    #if __STDC_VERSION__ > 201700L
18    printf("Standard-C17 oder neuer\n");
19    #elif __STDC_VERSION__ > 201100L
20    printf("Standard-C11\n");
21    #elif __STDC_VERSION__ > 199900L
22    printf("Standard-C99\n");
23    #elif __STDC_VERSION__ > 199400L
24    printf("Standard-C95\n");
25    #else
26    printf("Standard-C89 oder aelter\n");
27    #endif
28
29    return 0;
30 }
```

Dieses Beispielprogramm gibt folgendes aus (wenn es in der Quellcodedatei `kap10_05.c` gespeichert ist und am 04.08.2022 um 15:47:45 Uhr mit einem Standard-C11-Compiler compiliert wurde):

```
Programm wurde compiliert am Aug  4 2022 um 15:47:45.
Diese Programmzeile steht in Zeile 9 in der Datei kap10_05.c.
Standard-C-Compiler
Standard-C11
```

Auch werden für Compiler und Betriebssystem je eine Konstante vordefiniert. Die Konstanten der gängigen Compiler und Betriebssystem werden in den folgenden beiden Tabellen aufgelistet (die Liste ist schon etwas älter ...):

Vordefiniertes Makro	**Compiler**
`_MSC_VER`	Microsoft C ab Version 6.0
`_QC`	Microsoft Quick C ab Version 2.51
`__TURBOC__`	Borland Turbo C, Turbo C++ und BC++

`__BORLANDC__`	Borland C++
`__ZTC__`	Zortech C und C++
`__SC__`	Symantec C++
`__WATCOMC__`	WATCOM C
`__GNUC__`	GNU C
`__EMX__`	Emx GNU C

Vordefiniertes Makro	**Betriebssystem**
`__unix__`	UNIX-System
`__unix`	UNIX-System
`__MS_DOS__`	MS-DOS
`__WIN32__`	MS Windows ab 95
`__OS2__`	OS2
`_Windows`	Zielsystem MS Windows
`__NT__`	MS Windows NT
`__linux__`	Linux-System
`__FreeBSD__`	FreeBSD
`__OpenBSD__`	OpenBSD
`_SGI_SOURCE`	SGI-IRIX mit Extension *.sgi
`_MIPS_ISA`	SGI-IRIX
`__hpux`	HP-UX

Vordefinierte Makros können mit `#undef` nicht entfernt werden.

10.6. #undef

Der Präprozessor-Befehl `#undef` kann dazu genutzt werden, um ein Label bzw. ein Makro wieder zu entfernen. Dazu muss hinter dem Befehl der Name des Labels bzw. des Makros gesetzt werden. Die Syntax lautet also wie folgt:

`#undef` *`Name`*

Es wird kein Fehler verursacht, wenn dieser Befehl auf einen Namen angewendet wird, der gar nicht oder nicht mehr definiert ist.

Ist ein Label bzw. ein Makro erst einmal entfernt, kann anschließend ein neues Label bzw. ein neues Makro mit diesem Namen definiert werden.

10.7. #if

Ähnlich wie bei den Befehlen `#ifdef` und `#ifndef` kann mit dem Befehl `#if` eine bedingte Compilierung erreicht werden. Mit dem Befehl `#if` werden aber nicht Labels abgefragt, sondern es können beliebige Bedingungen verwendet werden. Die Bedingung muss einen konstanten arithmetischen Wert ergeben, der als Wahrheitswert interpretiert werden kann. Dieser Wert muss bereits beim Compilieren feststehen.

Der Bereich der bedingten Compilierung muss mit dem Befehl `#endif` (in einer eigenen Zeile) abgeschlossen werden. Es ist auch möglich, mit `#else` einen "Sonst"-Zweig in die bedingte Compilierung mit einzubringen.

10.8. #line

Mit dem Präprozessor-Befehl `#line` können Zeilennummer und Dateiname für die vordefinierten Makros `__LINE__` und `__FILE__` verändert werden. Dies kann bei der Fehlersuche sinnvoll sein, wenn die Quellcodedatei vor dem Compilieren von einem Tool bearbeitet wird, bei dem Zeilen eingefügt oder mehrere Quellcodedateien zu einer zusammen gefügt werden.

Dabei gibt es zwei Formen der Anwendung: In der ersten Form werden hinter dem Befehl die Zeilennummer und der Dateiname angegeben; der Dateiname muss in Anführungszeichen gesetzt werden. Diese Angaben gelten für die Quellcodezeile, die diesem Befehl folgt.

```
#line n "Dateiname"
```

In der zweiten Form wird der Dateiname weggelassen; dieser entspricht dann dem tatsächlichen Dateinamen bzw. dem angegebenen Dateinamen des vorherigen `#line`-Befehls.

```
#line n
```

Ältere Compiler erlauben auch eine verkürzte Schreibweise, die aber **vom Standard-C nicht akzeptiert** wird. Dabei wird das Wort `line` weggelassen:

```
# n "Dateiname"
```

Beispiel:

kap10_06.c

```
01 #include <stdio.h>
02
03 int main()
04 {
05    #line 12345 "Neuer_Dateiname.c"
06    printf("Diese Programmzeile steht in Zeile ");
07    printf("%d in der Datei %s.\n", __LINE__, __FILE__);
08
09    return 0;
10 }
```

Dieses Beispielprogramm gibt folgendes aus (egal in welcher Quellcodedatei es gespeichert ist):

```
Diese Programmzeile steht in Zeile 12346 in der Datei Neuer_Dateiname.c.
```

10.9. #-Operator

Der #-Operator ist ein unärer Präprozessor-Operator und wird in Makros verwendet. Während das Makro umgesetzt wird, wird der #-Operator mit dem dahinter stehenden Makro-Parameter ersetzt durch den aktuellen Wert des Makro-Parameters eingeschlossen in Anführungszeichen. Dadurch wird der Wert des Makro-Parameters, der hinter dem Operator steht, in eine Zeichenkette umgewandelt, egal um was für einen Datentypen es sich handelt. Während die Zeichenkette erzeugt wird, werden mehrere hintereinander stehende Weiße Leerzeichen ersetzt durch ein Leerzeichen. Anführungszeichen und Backslashs, die im Makro-Parameter enthalten sind, bekommen noch ein zusätzliches Backslash davor gesetzt, damit die ursprüngliche Bedeutung erhalten bleibt.

Beispiel:

```
#define Makro(a) printf(#a)
```

Beim Durchlauf des Präprozessors werden die Makroaufrufe

```
Makro(7);
Makro("\n");
```

werden ersetzt durch

```
printf("7");
printf("\"\\n\"");
```

Achtung:

Eine Reihe von älteren C-Compilern fügen vor Anführungszeichen und Backslashs kein weiteren Backslash ein. Dies ist aber im Standard-C nicht zulässig.

10.10. ##-Operator

Mit dem `##`-Operator lassen sich Tokens in einer Makrodefinition zusammenfügen. D.h. das Token vor und das Token nach dem Operator werden zu einem Token zusammengesetzt. Dabei wird für den zweiten Token der aktuelle Wert eingefügt, sofern dieser ein Makro-Parameter ist.

Im folgenden Beispiel setzt der Präprozessor die Makros `Makro(1)` und `Makro(2)` korrekt in die Variablen `temp1` und `temp2` um. Da der Präprozessor die Makros vor dem Compilieren auflöst, wird in der Schleife das Makro `Makro(i)` zur Variable `tempi` umgesetzt.

Beispiel:

kap10_07.c

```
01 #include <stdio.h>
02
03 #define MAKRO(i) temp ## i
04
05 int main()
06 {
07    int temp1, temp2, tempi, i;
08
09    MAKRO(1) = 5;              /* setzt die Variable temp1 auf 5    */
10    MAKRO(2) = 7;              /* setzt die Variable temp2 auf 7    */
11
12    for (i = 1; i <= 2; i++)
13       MAKRO(i) = 0;           /* setzt 2x die Variable tempi auf 0 */
14
15    printf("%i\n", MAKRO(1)); /* gibt temp1 (also 5) aus           */
16    printf("%i\n", MAKRO(2)); /* gibt temp2 (also 7) aus           */
17
18    return 0;
19 }
```

10.11. #pragma

Mit Hilfe des Präprozessor-Befehls `#pragma` können Informationen oder Einstellungen an den Compiler übermittelt werden. Diese Informationen und Einstellungen sind abhängig vom Compiler. Jeder Compiler sollte `#pragma`-Befehle ignorieren, wenn die Informationen und Einstellungen nicht zum Compiler passen. Dieser Befehl wurde mit dem Standard-C eingeführt.

Beispiel:

```
#pragma FENV_ACCESS ON
```

Dieser Befehl weist den Compiler an, beim Compilieren Überwachungen und Exceptions rund um die Fließkomma-Arithmetik mit einzubauen.

Mit C99 wurden dann noch die Standard-Pragmas eingeführt. Während im obigen Beispiel das Pragma Compiler-abhängig ist, wird im nächsten Beispiel das C99-Standard-Pragma (gleichen Namens) angewendet.

Beispiel:

```
#pragma STDC FENV_ACCESS ON
```

Es gibt im C99 nur die drei folgenden Standard-Pragmas. Diese können auf `ON`, `OFF` oder auf `DEFAULT` (Vorgabewert des Compilers) gesetzt werden.

```
#pragma STDC FP_CONTRACT ON
#pragma STDC FENV_ACCESS ON
#pragma STDC CS_LIMITED_RANGE ON
```

Die `#pragma`-Befehle können nur an zwei verschiedenen Stellen im Quelltext eingesetzt werden: Entweder in der obersten Ebene (Top Level) vor der ersten Deklaration oder innerhalb eines Blockes – auch hier vor der ersten Deklaration.

Im ersten Fall ist der Befehl bis zum Ende des Quelltextes oder bis zu einem erneuten Befehl mit dem gleichen Pragma gültig. Liegt der erneute Befehl innerhalb eines Blockes, so überlagert dieser erneute Befehl den ersten nur innerhalb des Blockes; nach dem Block ist wieder das erste Pragma gültig.

Im zweiten Fall ist der Befehl bis zum Ende des Blockes oder bis zu einem erneuten Befehl mit dem gleichen Pragma gültig.

Mit C99 wurde zusätzlich auch ein `_Pragma`-Operator eingeführt. Dieser wird während des Präprozessor-Laufes zusammen mit dem Parameter zu einem `#pragma`-Befehl umgewandet.

Beispiel:

Die Zeile

```
_Pragma("STDC FENV_ACCESS ON")
```

wird während des Präprozessor-Laufes umgewandelt in

```
#pragma STDC FENV_ACCESS ON
```

Im Gegensatz zum `#pragma`-Befehl kann der `_Pragma`-Operator durch Makros erstellt werden.

10.12. #error

Der Präprozessor-Befehl `#error` wurde erst mit dem Standard-C eingeführt. Der Befehl erzeugt eine Fehlermeldung während des Compilierens, d.h. das Programm kann dadurch nicht compiliert werden. Die Fehlermeldung, die ausgegeben werden soll, muss hinter dem Befehl angegeben werden.

Beispiel:

```
#ifndef Labelname
#error Labelname ist nicht definiert!
#endif
```

In diesem Beispiel wird während des Compilierens die Compiler-Fehlermeldung "Labelname ist nicht definiert!" ausgegeben, sofern `Labelname` nicht definiert ist. Damit kann der `#error`-Befehl verwendet werden, um z.B. Wiedersprüche in der Programmierung zu finden.

Beispiel:

```
#define MAX 255

#if (MAX % 256) != 0
#error "MAX muss ein Vielfaches von 256 sein!"
#endif
```

In diesem Beispiel ist die Fehlermeldung in Anführungszeichen gesetzt worden, damit das Token `MAX` nicht durch den `#define`-Befehl durch 255 ersetzt wird. Die Anführungszeichen werden in der Fehlermeldung mit ausgegeben.

11. Datei-Ein- und Ausgabe

Das Einlesen von Dateien und das Schreiben in Dateien wird im allgemeinen Datei-Ein- und Ausgabe genannt. Genau wie bei der Bildschirmausgabe und der Eingabe über Tastatur ist auch hier von Strömen die Rede, in die hineingeschrieben und/oder aus denen gelesen wird. Die allgemeine Vorgehensweise ist dabei immer gleich. Zuerst muss die Datei geöffnet werden, dann wird die Position gesucht, ab der entweder gelesen bzw. geschrieben wird, schließlich wird die eigentliche Aktion durchgeführt, nämlich das Lesen oder Schreiben der Daten und zum Schluss wird die Datei wieder geschlossen. Ganz ähnlich gehen Sie ja auch bei einer Textverarbeitung vor: Textdatei öffnen (Einlesen), Änderungen vornehmen, speichern (Schreiben) und wieder schließen.

Der Zugriff auf Dateien wird hier durch einen Zeiger auf einen Datenstrom durchgeführt; der Datentyp lautet `FILE *`. Verwendet wird er wie ein normaler Datentyp. Im folgenden Beispiel wird eine Zeigervariable namens `Datei` für einen Dateizugriff definiert (es können nur Zeiger verwendet werden!).

Beispiel:

```
FILE *Datei;
```

Mit dieser Anweisung ist noch kein Dateiname und kein Pfad spezifiziert, aber damit ist eine Zeiger-Variable geschaffen, die später auf den Datenstrom einer Datei zeigt. Die Spezifizierung von Dateiname und Pfad geschieht erst beim Öffnen einer Datei. Dabei wird der Datenstrom geöffnet und der zuvor definierte Zeiger auf `FILE` zeigt auf den Datenstrom. Anschließend wird bei jedem Dateizugriff dieser Datenstrom-Zeiger angegeben, um festzulegen, in welche Datei geschrieben bzw. aus welcher Datei gelesen werden soll.

11.1. Öffnen und Schließen von Dateien

Zum Öffnen einer Datei wird die `fopen`-Funktion verwendet. Nur hier werden Name und Pfad der Datei angegeben. Ferner muss der Modus (siehe Tabelle weiter unten) mitgeteilt werden, in dem die Datei geöffnet werden soll.

Bei der Angabe des Pfades können unter Windows die Backslashs wahlweise als doppelte Backslashs (`'\\'`) oder als einfache Slashs (`'/'`) angegeben werden; letzteres ist zu empfehlen, da hiermit die Kompatibilität zu Linux-Systemen erhalten bleibt.

Beispiel:

kap11_01.c

```
01 #include <stdio.h>
02
03 int main()
04 {
05    FILE *Datei;
06
07    Datei = fopen("U:/TEST.TXT", "rb");
08    if (Datei == NULL)
09       printf("Datei konnte nicht geöffnet werden!\n");
10    else
11    {
12       /*
13       ...                  Lesen bzw. Schreiben von Daten
14       */
15       fclose(Datei);    /* Datei schließen                */
16    }
17
18    return 0;
19 }
```

In diesem Programm wird - wie oben bereits erläutert - ein Zeiger auf den Datenstrom definiert. Dann wird die `fopen`-Funktion angewendet mit dem Dateinamen (inkl. Pfad) und dem Dateimodus als Parameter. Das Ergebnis ist ein Zeiger auf den Datenstrom dieser Datei und wird in der Variablen `Datei` gespeichert. Anschließend wird dieser Zeiger geprüft, ob er gleich dem `NULL`-Zeiger ist. Dies ist dann der Fall, wenn beim Öffnen der Datei ein Fehler aufgetreten ist und die Datei nicht geöffnet werden konnte (z.B. weil die Datei nicht existiert). Ist die Datei geöffnet, können Daten aus der Datei gelesen bzw. in die Datei geschrieben werden (genaueres dazu in den nächsten zwei Abschnitten). Zum Schluss wird die Datei mit der `fclose`-Funktion wieder geschlossen. Als Parameter wird hier der Zeiger auf den Datenstrom übergeben. Auch die `fclose`-Funktion hat ein Ergebnis, das gleich 0 ist, wenn beim Schließen der Datei kein Fehler aufgetreten ist, und sonst ungleich 0 ist. Hier wird aber dieses Ergebnis nicht verarbeitet.

Am Programmende werden alle geöffneten Dateien automatisch geschlossen, von daher wäre die `fclose`-Funktion nicht nötig. Das Weglassen der `fclose`-Funktion birgt aber in größeren Programmen eine Fehlerquelle und ist deshalb kein sauberer Programmierstil! Alternativ können alle noch offenen Dateien mit der `_fcloseall`-Funktion auf einen Schlag geschlossen werden (Aufruf: `_fcloseall();`).

Hier nun die Tabelle mit den verschiedenen Modi zum Öffnen von Dateien:

Modus	**Parameter**
Lesemodus (read)	`"r"`
Schreibmodus (write)	`"w"`
Anhängen (append)	`"a"`
Binärmodus (binary)	`"b"`
Textmodus (text)	`"t"`

Die Modi können auch kombiniert werden. Z.B.

Modus	**Parameter**
binäres Lesen	`"rb"`
Anhängen an Textdatei	`"at"`
Lesen und Schreiben einer Textdatei	`"rwt"`

11.2. Ausgabe in Dateien

Daten können formatiert oder unformatiert in Dateien geschrieben werden. Das formatierte Schreiben gleicht der Bildschirmausgabe. Der Befehl lautet hier `fprintf`; die komplette Syntax sieht wie folgt aus.

```
int fprintf(FILE *stream, char *format[, arguments]);
```

Der Unterschied gegenüber dem `printf`-Befehl besteht aus dem `"f"`, das vor den Befehl gesetzt wird und aus dem zusätzlichen Parameter, der den Datenstrom, also die Datei angibt. Die weiteren Parameter haben sich nicht verändert (siehe Kapitel *Datenausgabe auf dem Bildschirm*).

Das folgende Beispiel erzeugt eine neue Datei und schreibt 10 Zahlen in diese Datei hinein. Die Zahlen werden beim Schreiben in die Datei auch auf dem Bildschirm ausgegeben.

Beispiel:

kap11_02.c

```
01 #include <stdio.h>
02
03 int main()
04 {
05    FILE *Datei;
06    int i, Zahl[10] = { 7, 3, 9, 15, 4, 27, 98, 34, 85, 62};
07    char fname[] = "U:/TEST.TXT";
08
09    Datei = fopen(fname, "w");
```

```
10    if (Datei == NULL)
11       printf("Datei nicht erzeugt/geoeffnet!\n");
12    else
13    {
14       for (i = 0; i < 10; i++)
15       {
16          fprintf(Datei, "%i\n", Zahl[i]);
17          printf("Zahl %i: %i\n", i, Zahl[i]);
18       }
19       fclose(Datei);
20    }
21
22    return 0;
23 }
```

Für das unformatierte Schreiben gibt es den `fputc`-Befehl, der Daten zeichenweise in eine Datei schreibt. Dies ist gerade bei Texten oder bei Daten mit verschiedenen bzw. unbekannten Datentypen interessant. Die Syntax für diesen Befehl lautet wie folgt.

```
int fputc(int c, FILE *stream);
```

Es wird das zu schreibende Zeichen und der Zeiger auf den Datenstrom als Parameter angegeben. Der Datenstrom wird angegeben, um die Datei auszuwählen, in die das Zeichen geschrieben werden soll. Als Ergebnis dieser Funktion wird das geschriebene Zeichen zurückgegeben. Das Zeichen (Parameter und Rückgabewert) ist eine ganze Zahl (`int`), die als `unsigned char` interpretiert werden muss. Wird eine negative Zahl oder der Wert EOF zurückgegeben, ist ein Fehler aufgetreten. Ein Beispiel für das zeichenweise Schreiben ist am Ende des Kapitels.

11.3. Einlesen von Dateien

Auch das Einlesen aus Dateien kann formatiert oder unformatiert geschehen. Das formatierte Lesen geschieht analog zum formatierten Schreiben mit dem `fscanf`-Befehl. Die vollständige Syntax lautet:

```
int fscanf(FILE *stream, char *format[, arguments]);
```

Auch hier ist der Unterschied zum `scanf`-Befehl (Einlesen von der Tastatur) nur das vorne angehangende `"f"` und der Datenstrom als zusätzlicher Parameter. Alles andere ist identisch mit dem bereits bekannten `scanf`-Befehl (siehe Kapitel *Dateneingabe über die Tastatur*).

Das folgende Beispiel liest aus der Datei `U:\TEST.TXT`, die im vorigen Beispiel erzeugt wurde, die 10 Zahlen wieder ein und gibt diese auf dem Bildschirm aus.

Beispiel:

kap11_03.c

```
01 #include <stdio.h>
02
03 int main()
04 {
05    FILE *Quelle;
06    int i, Zahl[10];
07    char fname[] = "U:/TEST.TXT";
08
09    Quelle = fopen(fname, "r");
10    if (Quelle == NULL)
11       printf("Quelldatei nicht geoeffnet!\n");
12    else
13    {
14       for (i = 0; i < 10; i++)
15       {
```

```
16          fscanf(Quelle, "%i", &Zahl[i]);
17          printf("Zahl %i: %i\n", i, Zahl[i]);
18       }
19       fclose(Quelle);
20    }
21
22    return 0;
23 }
```

Für das unformatierte Einlesen gibt es den `fgetc`-Befehl, der die Datei zeichenweise einliest. Dies ist dann interessant, wenn der Aufbau der Datei nicht bekannt ist. Die Syntax für diesen Befehl lautet wie folgt.

```
int fgetc(FILE *stream);
```

Es wird nur der Zeiger auf den Datenstrom als Parameter angegeben, um die Datei auszuwählen, von der ein Zeichen gelesen werden soll. Als Ergebnis dieser Funktion wird das gelesene Zeichen zurückgegeben. Die Funktion gibt ein `int` zurück, das als `unsigned char` interpretiert werden muss. Wird eine negative Zahl – z.B. der Wert `EOF` (= -1) – zurückgegeben, ist das Dateiende erreicht oder es ist ein Fehler aufgetreten. Ein Beispiel für das zeichenweise Einlesen ist am Ende des Kapitels.

11.4. Zusammenfassung

Hier nun eine Zusammenfassung aller Befehle für die Datei-Ein- und Ausgabe. Alle diese Befehle benötigen die Headerdatei `stdio.h`. Danach wird noch einmal ein Beispiel gezeigt.

Befehl	**Funktionsprototyp**	**Funktionsergebnis**
Datei öffnen	`FILE *fopen(char *name, char *mode);`	`== NULL` wenn Fehler, sonst Zeiger auf Datenstrom
Zeichen lesen	`int fgetc(FILE *stream);`	`== EOF` wenn Fehler/Dateiende, sonst das gelesene Zeichen
Daten lesen	`int fscanf(FILE *stream, char *format[, arguments]);`	`== EOF` wenn Fehler/Dateiende, sonst Anzahl der gelesenen Felder
Zeichen schreiben	`int fputc(int c, FILE *stream);`	`== EOF` wenn Fehler, sonst das geschriebene Zeichen
Daten schreiben	`int fprintf(FILE *stream, char *format[, arguments]);`	`< 0` wenn Fehler, sonst Anzahl der geschriebenen Bytes
Dateiende prüfen	`int feof(FILE *stream);`	`!= 0` wenn Dateiende, `== 0` wenn kein Dateiende
Datei schließen	`int fclose(FILE *stream);`	`== EOF` wenn Fehler, `== 0` wenn kein Fehler
alle Dateien schließen	`int _fcloseall(void);`	`== EOF` wenn Fehler, sonst Anzahl der geschlossenen Dateien

Das folgende Beispiel kopiert die Datei `U:\TEST.TXT` nach `U:\TEST.BAK`, erzeugt also eine Sicherheitskopie der Originaldatei. Dabei wird die Originaldatei zeichenweise eingelesen und auch zeichenweise in die Zieldatei geschrieben.

Beispiel:

kap11_04.c

```
01 #include <stdio.h>
02
03 int main()
04 {
05    FILE *Quelle, *Ziel;
```

```
06    int i;
07    char fname1[] = "U:/TEST.TXT", fname2[] = "U:/TEST.BAK";
08
09    Quelle = fopen(fname1, "rb");
10    if (Quelle == NULL)
11       printf("Quelldatei nicht geoeffnet!\n");
12    else
13    {
14       Ziel = fopen(fname2, "wb");
15       if (Ziel == NULL)
16          printf("Zieldatei nicht geoeffnet!\n");
17       else
18       {
19          while ((i = fgetc(Quelle)) != EOF)
20             fputc(i, Ziel);
21          fclose(Ziel);
22       }
23       fclose(Quelle);
24    }
25
26    return 0;
27 }
```

12. Dynamische Speicherverwaltung

Bisher wurden nur Datentypen behandelt, deren Speicherplatzbedarf bereits zur Compilierungszeit feststanden und beim Erzeugen des Programms eingeplant werden konnten. Es ist jedoch nicht immer möglich, den Speicherbedarf exakt vorher zu planen, und es ist unökonomisch, jedesmal sicherheitshalber den maximalen Speicherplatz zu reservieren. C bietet daher die Möglichkeit, Speicherbereiche während des Programmlaufs zu reservieren, d.h. dem Programm zur Verfügung zu stellen. Wichtig ist, dass die reservierten Speicherbereiche spätestens zum Programmende wieder freigegeben werden.

12.1. Speicherbereiche reservieren

Speicherbereiche reservieren heißt, zur Laufzeit des Programms einen zusammenhängenden Speicherbereich dem Programm zugänglich zu machen und nach außen hin diesen Speicherbereich als belegt zu markieren. Dieser Speicherbereich liegt im sogenannten **Heap** (zu deutsch *Halde*), einem großen Speicherbereich, der vom Betriebssystem verwaltet wird.

Zum Reservieren von Speicherbereichen wird eine der beiden Funktionen `malloc` (steht für *Memory Allocation*) oder `calloc` (steht für *Cleared Memory Allocation*) verwendet. Dazu muss noch die Headerdatei `stdlib.h` oder `malloc.h` eingebunden werden. Die Funktionen sind wie folgt deklariert:

```
void *malloc(int Groesse);
void *calloc(int Anzahl_Elemente, int Groesse_eines_Elements);
```

Beide Funktionen liefern einen `void`-Zeiger auf den reservierten Speicherbereich (bei `calloc` wird der Speicher mit 0 initialisiert, bei `malloc` sind die Werte des Speicherbereichs undefiniert) oder den `NULL`-Zeiger, wenn nicht mehr genügend Speicher vorhanden ist. Wird beim Reservieren des Speichers ein Zeiger auf einen anderen Datentypen benötigt, wird der Ergebniszeiger von `malloc` bzw. `calloc` entsprechend implizit umgewandelt (siehe auch Typumwandlung). **Achtung: In C++ (d.h. auch bei Verwendung eines C++-Compilers für die Programmierung in C) muss er explizit umgewandelt werden!**

Beispiel:

kap12_01.c

```
01 #include <stdio.h>
02 #include <stdlib.h>
03
04 int main()
05 {
06    double *t = malloc(2 * sizeof(*t));
07
08    if (t != NULL)
09    {
10       *t = 3.1415296;
11       *(t + 1) = 2 * *t;
12       printf("    PI = %f\n", *t);
13       printf("2 * PI = %f\n", *(t + 1));
14
15       free(t);    /* Speicher wieder freigeben */
16    }
17    else
18       printf("Kein Speicher verfuegbar!\n");
19
20    return 0;
21 }
```

Zum Zeitpunkt der Compilierung wird nur der Platz für den Zeiger `t` eingeplant. Mit der Initialisierung des Zeigers wird Speicherplatz in der Größe von `2 * sizeof(*t)` Bytes (also `2 * sizeof(double)`

Bytes) zur Laufzeit des Programms bereitgestellt. Der Zeiger `t` zeigt anschließend auf diesen Speicherplatz (sofern Speicher vorhanden ist!).

Da Arrays und Zeiger intern identisch dargestellt und verarbeitet werden, lässt sich das oben angegebene Beispielprogramm auch mit Arrays schreiben.

Beispiel:

kap12_02.c

```
01 #include <stdio.h>
02 #include <stdlib.h>
03
04 int main()
05 {
06    double *t = malloc(2 * sizeof(*t));
07
08    if (t != NULL)
09    {
10       t[0] = 3.1415296;
11       t[1] = 2 * t[0];
12       printf("    PI = %f\n", t[0]);
13       printf("2 * PI = %f\n", t[1]);
14
15       free(t);   /* Speicher wieder freigeben */
16    }
17    else
18       printf("Kein Speicher verfuegbar!\n");
19
20    return 0;
21 }
```

Es lassen sich auch Speicherbereiche für Strukturen reservieren. Im folgenden Beispiel wird eine Struktur für imaginäre Zahlen definiert. Dann wird mit Hilfe der `calloc`-Funktion ein Speicherbereich für 2 Strukturen reserviert und dieser Bereich mit Werten gefüllt.

Beispiel:

kap12_03.c

```
01 #include <stdio.h>
02 #include <stdlib.h>
03
04 int main()
05 {
06    struct Imag
07    {
08       double Re;
09       double Im;
10    };
11    struct Imag *I = calloc(2, sizeof(*I));
12
13    if (I != NULL)
14    {
15       I->Re = 1;
16       I->Im = 0;
17       (I + 1)->Re = 0;
18       (I + 1)->Im = 1;
19       printf("  *I    = %f / %f\n", I->Re, I->Im);
20       printf("*(I+1) = %f / %f\n", (I + 1)->Re, (I + 1)->Im);
21
22       free(I);   /* Speicher wieder freigeben */
```

```
23     }
24     else
25        printf("Kein Speicher verfuegbar!\n");
26
27     return 0;
28  }
```

Die Größe des mit `malloc` bzw. `calloc` reservierten Speicherbereiches lässt sich mit der Funktion `realloc` vergrößern oder verkleinern. Dazu muss der Funktion ein Zeiger auf den bisherigen Speicherbereich und die neue Größe in Bytes angegeben werden. Je nach Compiler wird intern der reservierte Speicherbereich erweitert bzw. reduziert (sofern hinter dem bisher reservierten Speicherbereich noch genügend freier Heap vorhanden ist) oder es wird ein neuer Speicherbereich in der gewünschten Größe reserviert und der alte Speicherbereich anschließend freigegeben. Dabei bleiben die Daten vom alten Speicherbereich erhalten, d.h. ist der neue Speicherbereich größer, werden die Daten komplett in den neuen Speicherbereich kopiert, der restliche Speicherbereich wird nicht initialisiert. Ist dagegen der neue Speicherbereich kleiner, werden die Daten nur bis zur Größe des neuen Speicherbereichs kopiert, der Rest geht verloren.

Die Funktion ist folgendermaßen deklariert:

```
void *realloc(void *AlterSpeicherbereich, int NeueGroesse);
```

Beispiel:

kap12_04.c

```
01 #include <stdio.h>
02 #include <stdlib.h>
03
04 int main()
05 {
06    int *pArray = malloc(5 * sizeof(int));
07    int i;
08
09    if (pArray)
10    {
11       for (i = 0; i < 5; i++)
12       {
13          *(pArray + i) = i + 1;
14          printf("%i\n", *(pArray + i));
15       }
16
17       pArray = realloc(pArray, 1000 * sizeof(int));
18
19       if (pArray)
20       {
21          for (i = 0; i < 1000; i++)
22          {
23             *(pArray + i) = i + 1;
24             printf("%i\n", *(pArray + i));
25          }
26
27          free(pArray);
28       }
29    }
30
31    return 0;
32 }
```

Bei diesem Beispielprogramm wird erst Speicher für 5 Integerwerte reserviert, die anschließend in einer Schleife gesetzt und auf dem Bildschirm ausgegeben werden. Dann wird in Zeile 17 der reservierte

Speicherbereich erweitert auf 1000 Integerwerte, die genauso in einer Schleife gesetzt und auf dem Bildschirm ausgegeben werden. Wird die Zeile 17 auskommentiert, kommt es zur Laufzeit zu einem Speicherzugriffsfehler.

Besonderheiten der `realloc`-Funktion: Wird als neue Größe eine 0 angegeben, wird der alte Speicherbereich freigegeben; die Funktion `realloc` entspricht dann der Funktion `free` (siehe nächster Abschnitt). Wird als alter Speicherbereich der `NULL`-Zeiger angegeben, wird nur Speicher in der angegebenen Größe reserviert, d.h. die `realloc`-Funkion entspricht der `malloc`-Funktion.

12.2. Reservierte Speicherbereiche freigeben

Die `free`-Funktion (benötigt ebenfalls die Headerdatei `stdlib.h` oder `malloc.h`) gibt den reservierten Speicherbereich wieder frei, damit dieser von neuem belegt oder anderen Programmen zur Verfügung gestellt werden kann. Dazu wird der Zeiger, der auf den freizugebenden Speicherbereich zeigt, der Funktion als Parameter angegeben. Im folgenden Beispiel wird ein Speicherbereich reserviert und gleich wieder freigegeben.

Beispiel:

kap12_05.c

```
01 #include <stdio.h>
02 #include <stdlib.h>
03
04 int main()
05 {
06    void *z = malloc(1000); /* 1000 Bytes Speicher reservieren */
07
08    if (z != NULL)
09    {
10       printf("Speicher reserviert!\n");
11       free(z);              /* Speicher wieder freigeben       */
12    }
13    else
14       printf("Kein Speicher verfuegbar!\n");
15
16    return 0;
17 }
```

Nach dem Freigeben eines reservierten Speicherbereichs kann auf diesen nicht mehr zugegriffen werden!

12.3. Hinweise für die Verwendung von malloc, calloc und free

Einige Dinge sollten bei der dynamischen Speicherverwaltung beachtet werden, deren Missachtung oder Unkenntnis in manchen Fällen ein unvorhersehbares Programmverhalten bzw. einen Systemabsturz nach sich zieht, leider **ohne** Fehlermeldung vom Compiler oder vom Betriebssystem.

- Die `free`-Funktion darf **ausschließlich** Speicherbereiche freigeben, die zuvor mit `malloc`, `calloc` oder `realloc` reserviert wurden!
  ```
  int i, *ip1,*ip2;
  ip1 = malloc(sizeof(*ip1));
  ip2 = &i;
  free(ip1);  /* OK! */
  free(ip2);  /* Fehler!!! */
  ```
- Die `free`-Funktion darf jeden reservierten Speicherbereich nur einmal freigeben. Falls zwei oder mehr Zeiger auf den gleichen reservierten Speicherbereich zeigen, darf `free` nur mit einen dieser Zeiger aufgerufen werden. Die anderen Zeiger verweisen danach wohl noch auf den ehemals reservierten

Speicherbereich, dürfen aber nicht darauf zugreifen. Solche Zeiger werden dann **hängende Zeiger** (im englischen *dangling pointer*) genannt.

- `free(NULL);` bewirkt nichts; verursacht auch keinen Fehler.
- Reservierte Speicherbereiche unterliegen nicht den Gültigkeitsregeln von Variablen. D.h. sie existieren unabhängig von Blockgrenzen solange, bis sie wieder freigegeben werden oder das Programm beendet wird.
- Reservierte Speicherbereiche, auf die kein Zeiger mehr zeigt, sind nicht mehr zugänglich und werden **verwitwete Bereiche** genannt. Die englische Bezeichnung trifft das daraus resultierende Problem besser: **memory leak** (*Speicherleck*). Werden regelmäßig neue Speicherbereiche reserviert, ohne sie wieder freizugeben, bricht das Programm irgendwann wegen Speicherknappheit ab. Daher sollte gut darauf geachtet werden, nicht mehr benötigte Speicherbereiche wieder freizugeben.

Ein weiterer Grund für das Abstürzen von Programmen, die über lange Zeit laufen, liegt in der Zerstückelung des Heap durch ständiges Reservieren und Freigeben von Speicherbereichen. Mit Zerstückelung ist gemeint, dass sich kleinere belegte und freie Bereiche abwechseln, so dass das Reservieren eines größeren zusammenhängenden Speicherbereichs nicht mehr erfüllt werden kann, obwohl die Summe aller einzelnen freien Plätze ausreichen würde.

Dieses Problem verlangt ein Zusammenschieben aller belegten Plätze, so dass ein großer freier Bereich entsteht. Dies wird **garbage collection** (zu deutsch *Müllsammlung*) genannt. Die meisten C/C++-Compiler haben in ihrer Speicherverwaltung aus Effizienzgründen keine garbage collection eingebaut, weil sie nur in wenigen Fällen nötig ist und viel Rechenzeit benötigt.