C Programmierung

de.wikibooks.org

28. Dezember 2014

Inhaltsverzeichnis

0.1 Vorwort

Dieses Buch hat sich zum Ziel gesetzt, den Anwendern eine Einführung in C zu bieten, die noch keine oder eine geringe Programmiererfahrung haben. Es werden lediglich die grundlegenden Kenntnisse im Umgang mit dem Betriebssystem gefordert.

Allerdings soll auch nicht verschwiegen werden, dass das Lernen von C und auch das Programmieren in C viel Disziplin fordert. Die Sprache C wurde in den frühen 70er Jahren entwickelt, um das Betriebssystem UNIX nicht mehr in der fehleranfälligen Assemblersprache schreiben zu müssen. Die ersten Programmierer von C kannten sich sehr gut mit den Maschinen aus, auf denen sie programmierten. Deshalb, und aus Geschwindigkeitsgründen, verzichteten sie auf so manche Sprachmittel, mit denen Programmierfehler leichter erkannt werden können. Selbst die mehr als 30 Jahre, die seitdem vergangen sind, konnten viele dieser Fehler nicht ausbügeln, und so ist C mittlerweile eine recht komplizierte, fehleranfällige Programmiersprache. Trotzdem wird sie in sehr vielen Projekten eingesetzt, und vielleicht ist gerade das ja auch der Grund, warum Sie diese Sprache lernen möchten.

Wenn Sie wenig oder keine Programmiererfahrung haben, ist es sehr wahrscheinlich, dass Sie nicht alles auf Anhieb verstehen. Es ist sehr schwer, die Sprache C so zu erklären, dass nicht irgendwo vorgegriffen werden muss. Kehren Sie also hin und wieder zurück und versuchen Sie nicht, alles auf Anhieb zu verstehen. Wenn Sie am Ball bleiben, wird Ihnen im Laufe der Zeit vieles klarer werden.

Außerdem sei an dieser Stelle auf das Literatur- und Webverzeichnis[1] hingewiesen. Hier finden Sie weitere Informationen, die zum Nachschlagen, aber auch als weitere Einstiegshilfe gedacht sind.

Das besondere an diesem Buch ist aber zweifellos, dass es nach dem Wikiprinzip erstellt wurde. Das heißt, jeder kann Verbesserungen an diesem Buch vornehmen. Momentan finden fast jeden Tag irgendwelche Änderungen statt. Es lohnt sich also, hin und wieder vorbeizuschauen und nachzusehen, ob etwas verbessert wurde.

Auch Sie als Anfänger können dazu beitragen, dass das Buch immer weiter verbessert wird. Auf den Diskussionsseiten können Sie Verbesserungsvorschläge unterbreiten. Wenn Sie bereits ein Kenner von C sind, können Sie Änderungen oder Ergänzungen vornehmen. Mehr über das Wikiprinzip und Wikibooks erfahren sie im Wikibooks-Lehrbuch.

1 Kapitel 22.15 auf Seite 214

1 Grundlagen

1.1 Historisches

1964 begannen das Massachusetts Institute of Technology (MIT), General Electrics, Bell Laboratories und AT&T ein neues Betriebssystem mit der Bezeichnung Multics (Multiplexed Information and Computing Service) zu entwickeln. Multics sollte ganz neue Fähigkeiten wie beispielsweise Timesharing und die Verwendung von virtuellem Speicher besitzen. 1969 kamen die Bell Labs allerdings zu dem Schluss, dass das System zu teuer und die Entwicklungszeit zu lang wäre und stiegen aus dem Projekt aus.

Eine Gruppe unter der Leitung von Ken Thompson suchte nach einer Alternative. Zunächst entschied man sich dazu, das neue Betriebssystem auf einem PDP-7 von DEC (Digital Equipment Corporation) zu entwickeln. Multics wurde in PL/1 implementiert, was die Gruppe allerdings nicht als geeignet empfand, und deshalb das System in Assembler[1] entwickelte.

Assembler hat jedoch einige Nachteile: Die damit erstellten Programme sind z. B. nur auf einer Rechnerarchitektur lauffähig, die Entwicklung und vor allem die Wartung (also das Beheben von Programmfehlern und das Hinzufügen von neuen Funktionen) sind sehr aufwendig.

Deshalb suchte man für das System eine neue Sprache zur Systemprogrammierung. Zunächst entschied man sich für Fortran, entwickelte dann aber doch eine eigene Sprache mit dem Namen B, die stark beeinflusst von BCPL (Basic Combined Programming Language) war. Aus der Sprache B entstand dann die Sprache C. Die Sprache C unterschied sich von ihrer Vorgängersprache hauptsächlich darin, dass sie typisiert war. Später wurde auch der Kernel von Unix in C umgeschrieben. Auch heute noch sind die meisten Betriebssystemkerne, wie Windows oder Linux, in C geschrieben.

1978 schufen Dennis Ritchie und Brian Kernighan mit dem Buch *The C Programming Language*zunächst einen Quasi-Standard (auch als K&R-Standard bezeichnet). 1988 ist C erstmals durch das ANSI–Komitee standardisiert worden (als ANSI-C oder C-89 bezeichnet). Beim Standardisierungsprozess wurden viele Elemente der ursprünglichen Definition von K&R übernommen, aber auch einige neue Elemente hinzugefügt. Insbesondere Neuerungen der objektorientierten Sprache C++, die auf C aufbaut, flossen in den Standard ein.

Der Standard wurde 1999 überarbeitet und ergänzt (C99-Standard). Im Gegensatz zum C89-Standard, den praktisch alle verfügbaren Compiler beherrschen, setzt sich der C99-Standard nur langsam durch. Es gibt momentan noch kaum einen Compiler, der den neuen Standard vollständig unterstützt. Die meisten Neuerungen des C99-Standards sind im

1 `http://de.wikibooks.org/wiki/Assembler`

GNU-C-Compiler implementiert. Microsoft und Borland, die zu den wichtigsten Compilerherstellern zählen, unterstützen den neuen Standard allerdings bisher nicht, und es ist fraglich ob sie dies in Zukunft tun werden.

1.2 Was war / ist das Besondere an C

Die Entwickler der Programmiersprache legten größten Wert auf eine einfache Sprache mit maximaler Flexibilität und leichter Portierbarkeit auf andere Rechner. Dies wurde durch die Aufspaltung in den eigentlichen Sprachkern und die Programmbibliotheken (engl.: libraries) erreicht.

Daher müssen, je nach Bedarf, weitere Programmbibliotheken zusätzlich eingebunden werden. Diese kann man natürlich auch selbst erstellen um z. B. große Teile des eigenen Quellcodes thematisch zusammenzufassen, wodurch die Wiederverwendung des Programmcodes erleichtert wird.

Wegen der Nähe der Sprache C zur Hardware, einer vormals wichtigen Eigenschaft, um Unix leichter portierbar zu machen, ist C von Programmierern häufig auch als ein "Hochsprachen-Assembler" bezeichnet worden.

C selbst bietet in seiner *Standardbibliothek*nur rudimentäre Funktionen an. Die Standardbibliothek bietet hauptsächlich Funktionen für die Ein-/Ausgabe, Dateihandling, Zeichenkettenverarbeitung, Mathematik, Speicherreservierung und einiges mehr. Sämtliche Funktionen sind auf allen C-Compilern verfügbar. Jeder Compilerhersteller kann aber weitere Programmbibliotheken hinzufügen. Programme, die diese benutzen, sind dann allerdings nicht mehr portabel.

1.3 Der Compiler

Bevor ein Programm ausgeführt werden kann, muss es von einem Programm – dem Compiler – in Maschinensprache übersetzt werden. Dieser Vorgang wird als Kompilieren, oder schlicht als Übersetzen, bezeichnet. Die Maschinensprache besteht aus Befehlen (Folge von Binärzahlen), die vom Prozessor direkt verarbeitet werden können.

Neben dem Compiler werden für das Übersetzen des Quelltextes die folgenden Programme benötigt:

- Präprozessor
- Linker

Umgangssprachlich wird oft nicht nur der Compiler selbst als Compiler bezeichnet, sondern die Gesamtheit dieser Programme. Oft übernimmt tatsächlich nur ein Programm diese Aufgaben oder delegiert sie an die entsprechenden Spezialprogramme.

Vor der eigentlichen Übersetzung des Quelltextes wird dieser vom Präprozessor verarbeitet, dessen Resultat anschließend dem Compiler übergeben wird. Der Präprozessor ist im wesentlichen ein einfacher Textersetzer welcher Makroanweisungen auswertet und ersetzt

(diese beginnen mit #), und es auch durch Schalter erlaubt, nur bestimmte Teile des Quelltextes zu kompilieren.

Anschließend wird das Programm durch den Compiler in Maschinensprache übersetzt. Eine Objektdatei wird als Vorstufe eines ausführbaren Programms erzeugt. Einige Compiler - wie beispielsweise der GCC - rufen vor der Erstellung der Objektdatei zusätzlich noch einen externen Assembler auf. (Im Falle des GCC wird man davon aber nichts mitbekommen, da dies im Hintergrund geschieht.)

Der Linker (im deutschen Sprachraum auch häufig als Binder bezeichnet) verbindet schließlich noch die einzelnen Programmmodule miteinander. Als Ergebnis erhält man die ausführbare Datei. Unter Windows erkennt man diese an der Datei-Endung .exe.

Viele Compiler sind Bestandteil integrierter Entwicklungsumgebungen (IDEs, vom Englischen *Integrated Design Environment*oder *Integrated Development Environment*), die neben dem Compiler unter anderem über einen integrierten Editor verfügen. Wenn Sie ein Textverarbeitungsprogramm anstelle eines Editors verwenden, müssen Sie allerdings darauf achten, dass Sie den Quellcode im Textformat ohne Steuerzeichen abspeichern. Es empfiehlt sich, die Dateiendung .c zu verwenden, auch wenn dies bei den meisten Compilern nicht zwingend vorausgesetzt wird.

Wie Sie das Programm mit ihrem Compiler übersetzen, können Sie in der Referenz[2] nachlesen.

1.4 Hello World

Inzwischen ist es in der Literatur zur Programmierung schon fast Tradition, ein Hello World als einführendes Beispiel zu präsentieren. Es macht nichts anderes, als "Hello World" auf dem Bildschirm auszugeben, ist aber ein gutes Beispiel für die Syntax (Grammatik) der Sprache:

```
/* Das Hello-World-Programm */

#include <stdio.h>

int main()
{
  printf("Hello World!\n");

  return 0;
}
```

Dieses einfache Programm dient aber auch dazu, Sie mit der Compilerumgebung vertraut zu machen. Sie lernen

- Editieren einer Quelltextdatei
- Abspeichern des Quelltextes
- Aufrufen des Compilers und gegebenenfalls des Linkers
- Starten des compilierten Programms

2 Kapitel 22 auf Seite 157

Darüber hinaus kann man bei einem neu installierten Compiler überprüfen, ob die Installation korrekt war, und auch alle notwendigen Bibliotheken am richtigen Platz sind.

- In der ersten Zeile ist ein Kommentar zwischen den Zeichen /* und */ eingeschlossen. Alles, was sich zwischen diesen Zeichen befindet, wird vom Compiler nicht beachtet. Kommentare können sich über mehrere Zeilen erstrecken, dürfen aber nicht geschachtelt werden (obwohl einige Compiler dies zulassen).
- In der nächsten Zeile befindet sich die Präprozessor-Anweisung[3] `#include`. Der Präprozessor bearbeitet den Quellcode noch vor der Compilierung. An der Stelle der Include-Anweisung fügt er die (Header-)Datei `stdio.h`ein. Sie enthält wichtige Definitionen und Deklarationen für die Ein- und Ausgabeanweisungen[4]. [5]
- Das eigentliche Programm beginnt mit der Hauptfunktion `main`. Die Funktion `main`muss sich in jedem C-Programm befinden. Das Beispielprogramm besteht nur aus einer Funktion, Programme können aber in C auch aus mehreren Funktionen bestehen. In den runden Klammern können Parameter übergeben werden (später werden Sie noch mehr über Funktionen[6] erfahren).
  Die Funktion `main()`ist der Einstiegspunkt des C-Programms. `main()`wird immer sofort nach dem Programmstart aufgerufen.
- Die geschweiften Klammern kennzeichnen Beginn und Ende eines Blocks. Man nennt sie deshalb Blockklammern. Die Blockklammern dienen zur Untergliederung des Programms. Sie müssen auch immer um den Rumpf (Anweisungsteil) einer Funktion gesetzt werden, selbst wenn er leer ist.
- Zur Ausgabe von Texten wird die Funktion `printf`verwendet. Sie ist kein Bestandteil der Sprache C, sondern der Standard-C-Bibliothek[7] `stdio.h`, aus der sie beim Linken in das Programm eingebunden wird.
- Der auszugebende Text steht nach `printf`in Klammern. Die " zeigen an, dass es sich um reinen Text, und nicht um z. B. Programmieranweisungen handelt.
- In den Klammern steht auch noch ein `\n`. Das bedeutet einen Zeilenumbruch. Wann immer sie dieses Zeichen innerhalb einer Ausgabeanweisung schreiben, wird der Cursor beim Ausführen des Programms in eine neue Zeile springen.
- Über die Anweisung `return`wird ein Wert zurückgegeben. In diesem Fall geben wir einen Wert an das Betriebssystem zurück. Der Wert `0`teilt dem Betriebssystem mit, dass das Programm fehlerfrei ausgeführt worden ist.

C hat noch eine weitere Besonderheit: Klein- und Großbuchstaben werden unterschieden. Man bezeichnet eine solche Sprache auch als *case sensitive*. Die Anweisung `printf`darf also nicht als `Printf`geschrieben werden.

Hinweis: Wenn Sie von diesem Programm noch nicht viel verstehen, ist dies nicht weiter schlimm. Alle (wirklich alle) Elemente dieses Programms werden im Verlauf dieses Buches nochmals besprochen werden.

3 Kapitel 17 auf Seite 133
4 Kapitel 4 auf Seite 29
5 Unter Umständen besitzen Sie einen Compiler, der keine Headerdatei mit dem Namen `stdio`besitzt. Der C-Standard schreibt nämlich nicht vor, dass ein Header auch tatsächlich als Quelldatei vorliegen muss.
6 Kapitel 7 auf Seite 65
7 Kapitel 22.8 auf Seite 176

1.5 Ein zweites Beispiel: Rechnen in C

Wir wollen nun ein zweites Programm entwickeln, das einige einfache Berechnungen durchführt, und an dem wir einige Grundbegriffe lernen werden, auf die wir in diesem Buch immer wieder stoßen werden:

```
#include <stdio.h>

int main()
{
   printf("3 + 2 * 8 = %i\n", 3 + 2 * 8);
   printf("(3 + 2) * 8 = %i\n",(3 + 2) * 8);
   return 0;
}
```

Zunächst aber zur Erklärung des Programms: In Zeile 5 berechnet das Programm den Ausdruck `3 + 2 * 8`.Da C die Punkt-vor-Strich-Regel beachtet, ist die Ausgabe 19. Natürlich ist es auch möglich, mit Klammern zu rechnen, was in Zeile 6 geschieht. Das Ergebnis ist diesmal 40.

Das Programm besteht nun neben Funktionsaufrufen und der Präprozessoranweisung `#include`auch aus Operatoren und Operanden: Als *Operator*bezeichnet man Symbole, mit denen eine bestimmte Aktion durchgeführt wird, wie etwa das Addieren zweier Zahlen. Die Objekte, die mit den Operatoren verknüpft werden, bezeichnet man als *Operanden*. Bei der Berechnung von `(3 + 2) * 8`sind `+`, `*`und `( )`die Operatoren und `3`, `2`und `8`sind die Operanden. (%i ist eine Formatierungsanweisung die sagt, wie das Ergebnis als Zahl angezeigt werden soll, und ist nicht der nachfolgend erklärte Modulo-Operator.)

Keine Operatoren hingegen sind `{`, `}`, `"`, `;`, < und >. (Wobei < und > nur bei Verwendung in einem `#include`keine Operatoren sind. Außerhalb einer `#include`-Anweisung werden sie als Vergleichsoperatoren[8] verwendet.) Mit den öffnenden und schließenden Klammern wird ein Block eingeführt und wieder geschlossen, innerhalb der Anführungszeichen befindet sich eine Zeichenkette, mit dem Semikolon wird eine Anweisung abgeschlossen, und in den spitzen Klammern wird die Headerdatei angegeben.

Für die Grundrechenarten benutzt C die folgenden Operatoren:

Rechenart	**Operator**
Addition	+
Subtraktion	-
Multiplikation	*
Division	/
Modulo	%

Für weitere Rechenoperationen, wie beispielsweise Wurzel oder Winkelfunktionen, stellt C keine Funktionen zur Verfügung - sie werden aus Bibliotheken (Libraries) hinzugebunden. Diese werden wir aber erst später behandeln. Wichtig für Umsteiger: In C gibt es zwar den Operator ^, dieser stellt jedoch nicht den Potenzierungsoperator dar, sondern den bit-

8 Kapitel 22.6.4 auf Seite 165

weisen XOR-Operator! Für die Potenzierung muss deshalb ebenfalls auf eine Funktion der Standardbibliothek zurückgegriffen werden.

Häufig genutzt in der Programmierung wird auch der Modulo-Operator (%). Er ermittelt den Rest einer Division. Beispiel:

```
printf("Der Rest von 5 durch 3 ist: %i\n", 5 % 3);
```

Wie zu erwarten war, wird das Ergebnis 2 ausgegeben.

Wenn ein Operand durch 0 geteilt wird oder der Rest einer Division durch 0 ermittelt werden soll, so ist das Verhalten undefiniert. Das heißt, der ANSI-Standard legt das Verhalten nicht fest.

Ist das Verhalten nicht festgelegt, unterscheidet der Standard zwischen implementierungsabhängigem, unspezifiziertem und undefiniertem Verhalten:

- *Implementierungsabhängiges Verhalten*(engl. implementation defined behavior) bedeutet, dass das Ergebnis sich von Compiler zu Compiler unterscheidet. Allerdings ist das Verhalten nicht dem Zufall überlassen, sondern muss vom Compilerhersteller festgelegt und auch dokumentiert werden.
- Auch bei einem *unspezifizierten Verhalten*(engl. unspecified behavior) muss sich der Compilerhersteller für ein bestimmtes Verhalten entscheiden, im Unterschied zum implementierungsabhängigen Verhalten muss dieses aber nicht dokumentiert werden.
- Ist das Verhalten *undefiniert*(engl. undefined behaviour), bedeutet dies, dass sich nicht voraussagen lässt, welches Resultat eintritt. Das Programm kann bspw. die Division durch 0 ignorieren und ein nicht definiertes Resultat liefern, aber es ist genauso gut möglich, dass das Programm oder sogar der Rechner abstürzt oder Daten gelöscht werden.

Soll das Programm portabel sein, so muss man sich keine Gedanken darüber machen, unter welche Kategorie ein bestimmtes Verhalten fällt. Der C-Standard zwingt allerdings niemanden dazu, portable Programme zu schreiben, und es ist genauso möglich, Programme zu entwickeln, die nur auf einer Implementierung laufen. Nur im letzteren Fall ist für Sie wichtig zwischen implementierungsabhängigem, unspezifizierten und undefiniertem Verhalten zu unterscheiden.

1.6 Kommentare in C

Bei Programmen empfiehlt es sich, vor allem wenn sie eine gewisse Größe erreichen, diese zu kommentieren. Selbst wenn Sie das Programm übersichtlich gliedern, wird es für eine zweite Person schwierig werden, zu verstehen, welche Logik hinter Ihrem Programm steckt. Vor dem gleichen Problem stehen Sie aber auch, wenn Sie sich nach ein paar Wochen oder gar Monaten in Ihr eigenes Programm wieder einarbeiten müssen.

Fast alle Programmiersprachen besitzen deshalb eine Möglichkeit, Kommentare in den Programmtext einzufügen. Diese Kommentare bleiben vom Compiler unberücksichtigt. In C werden Kommentare in `/*`und `*/`eingeschlossen. Ein Kommentar darf sich über mehrere Zeilen erstrecken. Eine Schachtelung von Kommentaren ist nicht erlaubt.

In neuen C-Compilern, die den C99-Standard beherrschen, aber auch als Erweiterung in vielen C90-Compilern, sind auch einzeilige Kommentare, beginnend mit //zugelassen. Er wird mit dem Ende der Zeile abgeschlossen. Dieser Kommentartyp wurde mit C++ eingeführt und ist deshalb in der Regel auch auf allen Compilern verfügbar, die sowohl C als auch C++ beherrschen.

Beispiel für Kommentare:

```
/* Dieser Kommentar
   erstreckt sich
   über mehrere
   Zeilen */

#include <stdio.h>  // Dieser Kommentar endet am Zeilenende

int main()
{
  printf("Beispiel für Kommentare\n");
  //printf("Diese Zeile wird niemals ausgegeben\n");

  return 0;
}
```

Hinweis: Tipps zum sinnvollen Einsatz von Kommentaren finden Sie im Kapitel Programmierstil[9]. Um die Lesbarkeit zu verbessern, wird in diesem Wikibook häufig auf die Kommentierung verzichtet.

en:C Programming/History[10] es:Programación en C/Historia de C[11] fr:Programmation C/Introduction[12] it:C/Linguaggio/Panoramica[13] pt:Programar em C/História da linguagem C[14]

9 Kapitel 20 auf Seite 147
10 http://en.wikibooks.org/wiki/C%20Programming%2FHistory
11 http://es.wikibooks.org/wiki/Programaci%C3%B3n%20en%20C%2FHistoria%20de%20C
12 http://fr.wikibooks.org/wiki/Programmation%20C%2FIntroduction
13 http://it.wikibooks.org/wiki/C%2FLinguaggio%2FPanoramica
14 http://pt.wikibooks.org/wiki/Programar%20em%20C%2FHist%C3%B3ria%20da%20linguagem%20C

2 Variablen und Konstanten

2.1 Was sind Variablen?

Als nächstes wollen wir ein Programm entwickeln, das die Oberfläche A eines Quaders ermittelt. Bezeichnet man die Länge des Quaders mit a, die Breite mit b und die Höhe mit c, so gilt die Formel

$$A = 2 \cdot (a \cdot b + a \cdot c + b \cdot c)$$

Eine einmal eingeführte *Variable*, hier also a, b und auch c, ist in der Mathematik im weiteren Gang der Argumentation fest: sie ändert weder ihren Wert noch ihre Bedeutung.

Auch bei der Programmierung gibt es Variablen, diese werden dort allerdings anders verwendet als in der Mathematik: Eine Variable repräsentiert eine Speicherstelle, deren Inhalt während der gesamten Lebensdauer der Variable jederzeit verändert werden kann. Es ist so beispielsweise möglich, beliebig viele Quader nacheinander zu berechnen, ohne jedesmal neue Variablen einführen zu müssen.

Eine Variable kann bei der Programmierung also ihren *Wert*ändern. Jedoch zeugt es von schlechtem Programmierstil, im Verlauf des Quelltextes die *Bedeutung*einer Variablen zu ändern. Hat man also in einem Programm zur Kreisberechnung beispielsweise eine Variable namens *radius*, in der der Radius eines Kreises abgespeichert ist, so hüte man sich davor, in ihr etwa den Flächeninhalt desselben Kreises oder etwas völlig Anderes abzulegen. Der Quelltext würde dadurch erheblich weniger verständlich.

Weiteres zur Benennung von Variablen lese man im Abschnitt Namensgebung[1] nach.

Das Programm zur Berechnung einer Quaderoberfläche könnte etwa wie folgt aussehen:

```
#include <stdio.h>

int main(void)
{
  int a,b,c;

  printf("Bitte Länge des Quaders eingeben:\n");
  scanf("%d",&a);
  printf("Bitte Breite des Quaders eingeben:\n");
  scanf("%d",&b);
  printf("Bitte Höhe des Quaders eingeben:\n");
  scanf("%d",&c);
  printf("Quaderoberfläche:\n%d\n", 2 * (a * b + a * c + b * c));
  getchar();
```

1 Kapitel 20.4 auf Seite 148

```
  return 0;
}
```

- Bevor eine Variable in C benutzt werden kann, muss sie deklariert werden (Zeile 5). Das bedeutet, *Bezeichner*(Name der Variable) und (Daten-)*Typ*(hier `int`) müssen vom Programmierer festgelegt werden, dann kann der Computer entsprechenden Speicherplatz vergeben und die Variable auch adressieren (siehe später: C-Programmierung: Zeiger[2]). Im Beispielprogramm werden die Variablen a, b, und c als Integer (Ganzzahl) deklariert.
- Mit der Bibliotheksfunktion `scanf`können wir einen Wert von der Tastatur einlesen und in einer Variable speichern (mehr zur Anweisung `scanf`im nächsten Kapitel[3]).
- Der Befehl "getchar()" (Zeile 14) sorgt dafür, dass der Benutzer das Ergebnis überhaupt sieht. Denn würde "getchar()" weggelassen, würde das Programm so blitzschnell ausgeführt und anschließend beendet, dass der Benutzer die Ausgabe "Quaderoberfläche: ...(Ergebnis)" gar nicht sehen könnte. "getchar()" wartet einen Tastendruck des Benutzers ab, bis mit der Ausführung des Programms (eigentlich mit derjenigen der Funktion) fortgefahren wird. *Anmerkung: Wenn man C-Programme in einer Unix-Shell ausführt oder unter Windows im sog. DOS-Fenster, ist getchar() nicht notwendig.*
- Dieses Programm enthält keinen Code zur Fehlererkennung; d. h., wenn man hier statt der ganzen Zahlen etwas anderes oder auch gar nichts eingibt, passieren sehr komische Dinge. Hier geht es zunächst nur darum, die Funktionen zur Ein- und Ausgabe kennenzulernen. Wenn Sie eigene Programme schreiben, sollten Sie darauf achten, solche Fehler zu behandeln.

2.2 Deklaration, Definition und Initialisierung von Variablen

Bekanntlich werden im Arbeitsspeicher alle Daten über Adressen angesprochen. Man kann sich dies wie Hausnummern vorstellen: Jede Speicherzelle hat eine eindeutige Nummer, die zum Auffinden von gespeicherten Daten dient. Ein Programm wäre jedoch sehr unübersichtlich, wenn jede Variable mit der Adresse angesprochen werden würde. Deshalb werden anstelle von Adressen *Bezeichner*(Namen) verwendet. Der Compiler wandelt diese dann in die jeweilige Adresse um.

Neben dem Bezeichner einer Variable, muss der *Typ*[4] mit angegeben werden. Über den Typ kann der Compiler ermitteln, wie viel Speicher eine Variable im Arbeitsspeicher benötigt.

Der Typ sagt dem Compiler auch, wie er einen Wert im Speicher interpretieren muss. Bspw. unterscheidet sich in der Regel die interne Darstellung von Fließkommazahlen (Zahlen mit Nachkommastellen) und Ganzzahlen (Zahlen ohne Nachkommastellen), auch wenn der ANSI-C-Standard nichts darüber aussagt, wie diese implementiert sein müssen. Werden allerdings zwei Zahlen beispielsweise addiert, so unterscheidet sich dieser Vorgang bei Fließkommazahlen und Ganzzahlen aufgrund der unterschiedlichen internen Darstellung.

Bevor eine Variable benutzt werden kann, müssen dem Compiler der Typ und der Bezeichner mitgeteilt werden. Diesen Vorgang bezeichnet man als *Deklaration.*

2 Kapitel 9 auf Seite 77
3 Kapitel 4.2 auf Seite 33
4 Kapitel 22.6.10 auf Seite 174

Darüber hinaus muss Speicherplatz für die Variablen reserviert werden. Dies geschieht bei der *Definition*der Variable. Es werden dabei sowohl die Eigenschaften definiert als auch Speicherplatz reserviert. Während eine Deklaration mehrmals im Code vorkommen kann, darf eine Definition nur einmal im ganzen Programm vorkommen.

Hinweis

- **Deklaration**ist nur die Vergabe eines Namens und eines Typs für die Variable.
- **Definition**ist die Reservierung des Speicherplatzes.
- **Initialisierung**ist die Zuweisung eines ersten Wertes.

Die Literatur unterscheidet häufig nicht zwischen den Begriffen Definition und Deklaration und bezeichnet beides als Deklaration. Dies ist insofern richtig, da jede Definition gleichzeitig eine Deklaration ist (umgekehrt trifft dies allerdings nicht zu: Nicht jede Deklaration ist eine Definition). Zur besseren Abgrenzung der beiden Begriffe verwenden wir den Oberbegriff *Vereinbarung*, wenn sowohl Deklaration wie auch Definition gemeint ist.

Beispiel:

```
int i;
```

Damit wird eine Variable mit dem Bezeichner `i`und dem Typ `int`(Integer) definiert. Es wird eine Variable des Typs Integer und dem Bezeichner i vereinbart sowie Speicherplatz reserviert (da jede Definition gleichzeitig eine Deklaration ist, handelt es sich hierbei auch um eine Deklaration). Mit

```
extern char a;
```

wird eine Variable deklariert. Das Schlüsselwort *extern*in obigem Beispiel besagt, dass die Definition der Variablen a irgendwo in einem anderen Modul des Programms liegt. So deklariert man Variablen, die später beim Binden (Linken) aufgelöst werden. Da in diesem Fall kein Speicherplatz reserviert wurde, handelt es sich um keine Definition. Der Speicherplatz wird erst über

```
 char a;
```

reserviert, was in irgendeinem anderen Quelltextmodul erfolgen muss.

Noch ein Hinweis: Die Trennung von Definition und Deklaration wird hauptsächlich dazu verwendet, Quellcode in verschiedene Module unterzubringen. Bei Programmen, die nur aus einer Quelldatei bestehen, ist es in der Regel nicht erforderlich, Definition und Deklaration voneinander zu trennen. Vielmehr werden die Variablen einmalig vor Gebrauch definiert, wie Sie es im Beispiel aus dem letzten Kapitel gesehen haben.

Für die Vereinbarung von Variablen müssen Sie folgende Regeln beachten:

Variablen mit unterschiedlichen Namen, aber gleichen Typs können in derselben Zeile deklariert werden. Beispiel:

```
int a,b,c;
```

Definiert die Variablen `int a`, `int b`und `int c`.

Nicht erlaubt ist aber die Vereinbarung von Variablen unterschiedlichen Typs und Namen in einer Anweisung wie etwa im folgenden:

```
float a, int b; /* Falsch */
```

Diese Beispieldefinition erzeugt einen Fehler. Richtig dagegen ist, die Definitionen von `float`und `int`mit einem Semikolon zu trennen, wobei man jedoch zur besseren Lesbarkeit für jeden Typen eine neue Zeile nehmen sollte:

```
float a; int b;
```

Auch bei Bezeichnern unterscheidet C zwischen Groß- und Kleinschreibung. So können die Bezeichner `name`, `Name`und `NAME`für unterschiedliche Variablen oder Funktionen stehen. Üblicherweise werden Variablenbezeichner klein geschrieben, woran sich auch dieses Wikibuch hält.

Für vom Programmierer vereinbarte Bezeichner gelten außerdem folgende Regeln:

- Sie müssen mit einem Buchstaben oder einem Unterstrich beginnen; falsch wäre z. B. `1_Breite`.
- Sie dürfen nur Buchstaben des englischen Alphabets (also keine Umlaute oder 'ß'), Zahlen und den Unterstrich enthalten.
- Sie dürfen nicht einem C-Schlüsselwort[5] wie z. B. `int`oder `extern`entsprechen.

Nachdem eine Variable definiert wurde, hat sie keinen bestimmten Wert, sondern besitzt lediglich den Inhalt, der sich zufällig in der Speicherzelle befunden hat (auch als "Speichermüll" bezeichnet). Einen Wert erhält sie erst, wenn dieser ihr zugewiesen wird, z. B: mit der Eingabeanweisung `scanf`. Man kann der Variablen auch direkt einen Wert zuweisen. Beispiel:

```
a = 'b';
```

oder

```
summe = summe + zahl;
```

Verwechseln Sie nicht den Zuweisungsoperator in C mit dem Gleichheitszeichen in der Mathematik. Das Gleichheitszeichen sagt aus, dass auf der rechten Seite das Gleiche steht wie auf der linken Seite. Der Zuweisungsoperator dient hingegen dazu, der linksstehenden Variablen den Wert des rechtsstehenden Ausdrucks zuzuweisen.

Die zweite Zuweisung kann auch wesentlich kürzer wie folgt geschrieben werden:

```
summe += zahl;
```

Diese Schreibweise lässt sich auch auf die Subtraktion (-=), die Multiplikation (*=), die Division (/=) und den Modulooperator (%=) und weitere Operatoren übertragen.

5 Kapitel 22.4 auf Seite 161

Einer Variablen kann aber auch unmittelbar bei ihrer Definition ein Wert zugewiesen werden. Man bezeichnet dies als *Initialisierung*. Im folgenden Beispiel wird eine Variable mit dem Bezeichner `a`des Typs `char`(character) deklariert und ihr der Wert 'b' zugewiesen:

```
char a = 'b';
```

Wenn eine Variable als extern deklariert und dabei gleichzeitig mit einem Wert initialisiert wird, dann ignoriert der Compiler das Schlüsselwort `extern`. So handelt es sich bspw. bei

```
extern int c = 10;
```

nicht um eine Deklaration, sondern um eine Definition. `extern`sollte deshalb in diesem Falle weg gelassen werden.

2.3 Ganzzahlen

Ganzzahlen sind Zahlen ohne Nachkommastellen. In C gibt es folgende Typen für Ganzzahlen:

- `char`(character): 1 Byte [6] bzw. 1 Zeichen (kann zur Darstellung von Ganzzahlen oder Zeichen genutzt werden)
- `short int`(integer): ganzzahliger Wert
- `int`(integer): ganzzahliger Wert

Bei der Vereinbarung wird auch festgelegt, ob eine ganzzahlige Variable vorzeichenbehaftet sein soll. Wenn eine Variable ohne Vorzeichen vereinbart werden soll, so muss ihr das Schlüsselwort `unsigned`vorangestellt werden. Beispielsweise wird über

```
unsigned short int a;
```

eine vorzeichenlose Variable des Typs `unsigned short int`definiert. Der Typ `signed short int`liefert Werte von mindestens -32.768 bis 32.767. Variablen des Typs `unsigned short int`können nur nicht-negative Werte speichern. Der Wertebereich wird natürlich nicht größer, vielmehr verschiebt er sich und liegt im Bereich von 0 bis 65.535. [7]

Wenn eine Integervariable nicht explizit als vorzeichenbehaftet oder vorzeichenlos vereinbart wurde, ist sie immer vorzeichenbehaftet. So entspricht beispielsweise

```
int a;
```

6 Der C-Standard sagt nichts darüber aus, wie breit ein Byte ist. Es ist nur festgelegt, dass ein Byte mindestens 8 Bit hat. Ein Byte kann aber auch beispielsweise 10 oder 12 Bit groß sein. Allerdings ist dies nur von Interesse, wenn Sie Programme entwickeln wollen, die wirklich auf jedem auch noch so exotischen Rechner laufen sollen.

7 Wenn Sie nachgerechnet haben, ist Ihnen vermutlich aufgefallen, dass 32.768 + 32.767 nur 65.534 ergibt, und nicht 65.535, wie man vielleicht vermuten könnte. Das liegt daran, dass der Standard nichts darüber aussagt, wie negative Zahlen intern im Rechner dargestellt werden. Werden negative Zahlen beispielsweise im Einerkomplement ^{http://de.wikipedia.org/wiki/Einerkomplement%20} gespeichert, gibt es zwei Möglichkeiten, die 0 darzustellen, und der Wertebereich verringert sich damit um eins. Verwendet die Maschine (etwa der PC) das Zweierkomplement ^{http://de.wikipedia.org/wiki/Zweierkomplement%20} zur Darstellung von negativen Zahlen, liegt der Wertebereich zwischen –32.768 und +32.767.

der Definition

```
signed int a;
```

Leider ist die Vorzeichenregel beim Datentyp `char`etwas komplizierter:

- Wird `char`dazu verwendet einen numerischen Wert zu speichern und die Variable nicht explizit als vorzeichenbehaftet oder vorzeichenlos vereinbart, dann ist es implementierungsabhängig, ob `char`vorzeichenbehaftet ist oder nicht.
- Wenn ein Zeichen gespeichert wird, so garantiert der Standard, dass der gespeicherte Wert der nichtnegativen Codierung im Zeichensatz entspricht.

Was versteht man unter dem letzten Punkt? Ein Zeichensatz hat die Aufgabe, einem Zeichen einen bestimmten Wert zuzuordnen, da der Rechner selbst nur in der Lage ist, Dualzahlen zu speichern. Im ASCII-Zeichensatz wird beispielsweise das Zeichen 'M' als 77 Dezimal bzw. 1001101 Dual gespeichert. Man könnte nun auch auf die Idee kommen, anstelle von

```
char c = 'M';
```

besser

```
char c = 77;
```

zu benutzen. Allerdings sagt der C-Standard nichts über den verwendeten Zeichensatz aus. Wird nun beispielsweise der EBCDIC-Zeichensatz verwendet, so wird aus 'M' auf einmal eine öffnende Klammer (siehe Ausschnitt aus der ASCII- und EBCDIC-Zeichensatztabelle rechts).

ASCII	**EBCDIC**	**Dezimal**	**Binär**
L	<	76	1001100
M	(	77	1001101
N	+	78	1001110
...	...	...	...

Man mag dem entgegnen, dass heute hauptsächlich der ASCII-Zeichensatz verwendet wird. Allerdings werden es die meisten Programmierer dennoch als schlechten Stil ansehen, den codierten Wert anstelle des Zeichens der Variable zuzuweisen, da nicht erkennbar ist, um welches Zeichen es sich handelt, und man vermutet, dass im nachfolgenden Programm mit der Variablen `c`gerechnet werden soll.

Für Berechnungen werden Variablen des Typs Character sowieso nur selten benutzt, da dieser nur einen sehr kleinen Wertebereich besitzt: Er kann nur Werte zwischen -128 und +127 (vorzeichenbehaftet) bzw. 0 bis 255 (vorzeichenlos) annehmen (auf einigen Implementierungen aber auch größere Werte). Für die Speicherung von Ganzzahlen wird deshalb der Typ Integer (zu deutsch: Ganzzahl) verwendet. Es existieren zwei Varianten dieses Typs: Der Typ `short int`ist mindestens 16 Bit breit, der Typ `long int`mindestens 32 Bit. Eine Variable kann auch als `int`(also ohne ein vorangestelltes `short`oder `long`) deklariert werden. In diesem Fall schreibt der Standard vor, dass der Typ `int`eine "natürliche Größe" besitzen soll. Eine solche natürliche Größe ist beispielsweise bei einem IA-32 PC (Intel-Architektur mit 32 Bit) mit Windows XP oder Linux 32 Bit. Auf einem 16-Bit-Betriebssystem wie etwa

MS-DOS beträgt die Größe 16 Bit. Auf anderen Systemen kann `int`aber auch eine andere Größe annehmen. Das Stichwort hierzu lautet Wortbreite[8].

Mit dem C99-Standard wurde außerdem der Typ `long long int`eingeführt. Er ist mindestens 64 Bit breit. Allerdings wird er noch nicht von allen Compilern unterstützt.

Die jeweiligen Variablentypen können den folgenden Wertebereich annehmen:

Typ	**Vorzeichenbehaftet**	**Vorzeichenlos**
char	-128 bis 127	0 bis 255
short int	-32.768 bis 32.767	0 bis 65.535
long int	-2.147.483.648 bis 2.147.483.647	0 bis 4.294.967.295
long long int	-9.223.372.036.854.775.808 bis 9.223.372.036.854.775.807	0 bis 18.446.744.073.709.551.615

Die Angaben sind jeweils Mindestgrößen. In einer Implementierung können die Werte auch noch größer sein. Die tatsächliche Größe eines Typs ist in der Headerdatei `<limits.h>`[9]abgelegt. `INT_MAX`ersetzt der Präprozessor beispielsweise durch den Wert, den der Typ `int`maximal annehmen kann (Voraussetzung ist allerdings, dass Sie durch `#include <limits.h>`die Headerdatei `limits.h`einbinden).

2.4 Erweiterte Zeichensätze

Wie man sich leicht vorstellen kann, ist der "Platz" für verschiedene Zeichen mit einem einzelnen Byte sehr begrenzt, wenn man bedenkt, dass sich die Zeichensätze verschiedener Sprachen unterscheiden. Reicht der Platz für die europäischen Schriftarten noch aus, gibt es für asiatische Schriften wie Chinesisch oder Japanisch keine Möglichkeit mehr, die vielen Zeichen mit einem Byte darzustellen. Bei der Überarbeitung des C-Standards 1994 wurde deshalb das Konzept eines *breiten Zeichens*(engl. wide character) eingeführt, das auch Zeichensätze aufnehmen kann, die mehr als 1 Byte für die Codierung eines Zeichen benötigen (beispielsweise Unicode-Zeichen). Ein solches "breites Zeichen" wird in einer Variable des Typs `wchar_t`gespeichert.

Soll ein Zeichen oder eine Zeichenkette (mit denen wir uns später noch intensiver beschäftigen werden) einer Variablen vom Typ `char`zugewiesen werden, so sieht dies wie folgt aus:

```
char c = 'M';
char s[] = "Eine kurze Zeichenkette";
```

Wenn wir allerdings ein Zeichen oder eine Zeichenkette zuweisen oder initialisieren wollen, die aus breiten Zeichen besteht, so müssen wir dies dem Compiler mitteilen, indem wir das Präfix `L`benutzen:

```
wchar_t c = L'M';
wchar_t s[] = L"Eine kurze Zeichenkette" ;
```

8 http://de.wikipedia.org/wiki/Wortbreite%20

9 Kapitel 22.10.2 auf Seite 177

Leider hat die Benutzung von `wchar_t`noch einen weiteren Haken: Alle Bibliotheksfunktionen, die mit Zeichenketten arbeiten, können nicht mehr weiterverwendet werden. Allerdings besitzt die Standardbibliothek für jede Zeichenkettenfunktion entsprechende äquivalente Funktionen, die mit `wchar_t`zusammenarbeiten: Im Fall von `printf`ist dies beispielsweise `wprintf`.

2.5 Kodierung von Zeichenketten

Eine Zeichenkette kann mit normalen ASCII-Zeichen des Editors gefüllt werden. Z. B. : `char s []="Hallo Welt";`. Häufig möchte man Zeichen in die Zeichenkette einfügen, die nicht mit dem Editor darstellbar sind. Am häufigsten ist das wohl die Nächste Zeile (engl. linefeed) und der Wagenrücklauf (engl. carriage return). Für diese Zeichen gibt es keine Buchstaben, wohl aber ASCII-Codes. Hierfür gibt es bei C-Compilern spezielle Schreibweisen:

ESCAPE-Sequencen		
Schreibweise	**ASCII-Nr.**	**Beschreibung**
\n	10	Zeilenvorschub (new line)
\r	13	Wagenrücklauf (carriage return)
\t	09	Tabulator
\b	08	Backspace
\a	07	Alarmton
\’	39	Apostroph
\”	34	Anführungszeichen
\\	92	Backslash-Zeichen
\o1o2o3		ASCII-Zeichen mit Oktal-Code
\xh1h2h3...hn		ASCII-Zeichen mit Hexadez.

2.6 Fließkommazahlen

Fließkommazahlen (auch als Gleitkomma- oder Gleitpunktzahlen bezeichnet) sind Zahlen mit Nachkommastellen. Der C-Standard kennt die folgenden drei Fließkommatypen:

- Den Typ `float`für Zahlen mit einfacher Genauigkeit.
- Den Typ `double`für Fließkommazahlen mit doppelter Genauigkeit.
- Der Typ `long double`für zusätzliche Genauigkeit.

Wie die Fließkommazahlen intern im Rechner dargestellt werden, darüber sagt der C-Standard nichts aus. Welchen Wertebereich ein Fließkommazahltyp auf einer Implementierung einnimmt, kann allerdings über die Headerdatei `float.h`[10]ermittelt werden.

Im Gegensatz zu Ganzzahlen gibt es bei den Fließkommazahlen keinen Unterschied zwischen vorzeichenbehafteten und vorzeichenlosen Zahlen. Alle Fließkommazahlen sind in C immer vorzeichenbehaftet.

10 Kapitel 22.10.2 auf Seite 177

Beachten Sie, dass Zahlen mit Nachkommastellen in US-amerikanischer Schreibweise dargestellt werden müssen. So muss beispielsweise für die Zahl `5,353`die Schreibweise `5.353`benutzt werden.

2.7 Speicherbedarf einer Variable ermitteln

Mit dem `sizeof`-Operator kann die Länge eines Typs auf einem System ermittelt werden. Im folgenden Beispiel soll der Speicherbedarf in Byte des Typs `int`ausgegeben werden:

```
#include <stdio.h>

int main(void)
{
  int x;

  printf("Der Typ int hat auf diesem System die Größe %i Byte.\n", sizeof(int));
  printf("Die Variable x hat auf diesem System die Größe %i Byte.\n", sizeof x);
  return 0;
}
```

Nach dem Ausführen des Programms erhält man die folgende Ausgabe:

```
Der Typ int hat auf diesem System die Größe 4 Byte.
Die Variable x hat auf diesem System die Größe 4 Byte.
```

Die Ausgabe kann sich auf einem anderen System unterscheiden, je nachdem, wie breit der Typ `int`ist. In diesem Fall ist der Typ 4 Byte lang. Wie viel Speicherplatz ein Variablentyp besitzt, ist implementierungsabhängig. Der Standard legt nur fest, dass `sizeof(char)`immer den Wert 1 ergeben muss.

Beachten Sie, dass es sich bei `sizeof`um keine Funktion, sondern tatsächlich um einen Operator handelt. Dies hat unter anderem zur Folge, dass keine Headerdatei eingebunden werden muss, wie dies bei einer Funktion der Fall wäre. Die in das Beispielprogramm eingebundene Headerdatei `<stdio.h>`wird nur für die Bibliotheksfunktion `printf`benötigt.

Der `sizeof`-Operator wird häufig dazu verwendet, um Programme zu schreiben, die auf andere Plattformen portierbar sind. Beispiele werden Sie im Rahmen dieses Wikibuches noch kennenlernen.

2.8 Konstanten

2.8.1 Symbolische Konstanten

Im Gegensatz zu Variablen, können sich konstante Werte während ihrer gesamten Lebensdauer nicht ändern. Dies kann etwa dann sinnvoll sein, wenn Konstanten am Anfang des Programms definiert werden, um sie dann nur an einer Stelle im Quellcode anpassen zu müssen.

Ein Beispiel hierfür ist etwa die Mehrwertsteuer. Wird sie erhöht oder gesenkt, so muss sie nur an einer Stelle des Programms geändert werden. Um einen bewussten oder unbewussten Fehler des Programmierers zu vermeiden, verhindert der Compiler, dass der Konstante ein neuer Wert zugewiesen werden kann.

In der ursprünglichen Sprachdefinition von Dennis Ritchie und Brian Kernighan (*K&R*) gab es nur die Möglichkeit, mit Hilfe des Präprozessors symbolische Konstanten zu definieren. Dazu dient die Präprozessoranweisung `#define`. Sie hat die folgende Syntax:

```
#define IDENTIFIER token-sequence
```

Bitte beachten Sie, dass Präprozessoranweisungen nicht mit einem Semikolon abgeschlossen werden.

Durch die Anweisung

```
#define MWST 19
```

wird jede vorkommende Zeichenkette MWST durch die Zahl 19 ersetzt. Eine Ausnahme besteht lediglich bei Zeichenketten, die durch Anführungszeichen oder Hochkommata eingeschlossen sind, wie etwa der Ausdruck

```
  "Die aktuelle MWST"
```

Hierbei wird die Zeichenkette MWST **nicht**ersetzt.

Die Großschreibung ist nicht vom Standard vorgeschrieben. Es ist kein Fehler, anstelle von `MWST`die Konstante `MwSt`oder `mwst`zu benennen. Allerdings benutzen die meisten Programmierer Großbuchstaben für symbolische Konstanten. Dieses Wikibuch hält sich ebenfalls an diese Konvention (auch die symbolischen Konstanten der Standardbibliothek werden in Großbuchstaben geschrieben).

ACHTUNG: Das Arbeiten mit `define`kann auch fehlschlagen: Da `define`lediglich ein einfaches Suchen-und-Ersetzen durch den Präprozessor bewirkt, wird folgender Code nicht das gewünschte Ergebnis liefern:

```
#include <stdio.h>

#define quadrat(x)  x*x // fehlerhaftes Quadrat implementiert

int main (int argc, char *argv [])
{
  printf ("Das Quadrat von 2+3 ist %d\n", quadrat(2+3));

  return 0;
}
```

Wenn Sie dieses Programm laufen lassen, wird es Ihnen sagen, dass das Quadrat von 2+3 = 11 sei. Die Ursache dafür liegt darin, dass der Präprozessor `quadrat(2+3)`durch `2+3 * 2+3`ersetzt.

Da sich der Compiler an die Regel Punkt-vor-Strich-Rechnung hält, ist das Ergebnis falsch. In diesen Fall kann man das Programm wie folgt modifizieren damit es richtig rechnet:

```
#include <stdio.h>

#define quadrat(x) ((x)*(x)) // richtige Quadrat-Implementierung

int main(int argc,char *argv[])
{
  printf("Das Quadrat von 2+3 ist %d\n",quadrat(2+3));

  return 0;
}
```

2.8.2 Konstanten mit `const`definieren

Der Nachteil der Definition von Konstanten mit `define`ist, dass dem Compiler der Typ der Konstante nicht bekannt ist. Dies kann zu Fehlern führen, die erst zur Laufzeit des Programms entdeckt werden. Mit dem ANSI-Standard wurde deshalb die Möglichkeit von C++ übernommen, eine Konstante mit dem Schlüsselwort `const`zu deklarieren. Im Unterschied zu einer Konstanten, die über `define`definiert wurde, kann eine Konstante, die mit `const`deklariert wurde, bei älteren Compilern Speicherplatz wie Variablen auch verbrauchen. Bei neueren Compilern wie GCC 4.3 ist die Variante mit `const`immer vorzuziehen, da sie dem Compiler ein besseres Optimieren des Codes erlaubt und die Kompiliergeschwindigkeit erhöht. Beispiel:

```
#include <stdio.h>

int main()
{
  const double pi = 3.14159;
  double d;

  printf("Bitte geben Sie den Durchmesser ein:\n");
  scanf("%lf", &d);
  printf("Umfang des Kreises: %lf\n", d * pi);
  pi = 5; /* Fehler! */
  return 0;
}
```

In Zeile 5 wird die Konstante `pi`deklariert. Ihr muss sofort ein Wert zugewiesen werden, ansonsten gibt der Compiler eine Fehlermeldung aus.

Damit das Programm richtig übersetzt wird, muss Zeile 11 entfernt werden, da dort versucht wird, der Konstanten einen neuen Wert zuzuweisen. Durch das Schlüsselwort `const`wird allerdings der Compiler damit beauftragt, genau dies zu verhindern.

2.9 Sichtbarkeit und Lebensdauer von Variablen

In früheren Standards von C musste eine Variable immer am Anfang eines Anweisungsblocks vereinbart werden. Seit dem C99-Standard ist dies nicht mehr unbedingt notwendig: Es reicht aus, die Variable unmittelbar vor der ersten Benutzung zu vereinbaren.[11]

11 Beim verbreiteten Compiler GCC muss man hierfür explizit Parameter `-std=c99`übergeben

Ein Anweisungsblock kann eine Funktion[12], eine Schleife[13] oder einfach nur ein durch geschwungene Klammern begrenzter Block von Anweisungen sein. Eine Variable lebt immer bis zum Ende des Anweisungsblocks, in dem sie deklariert wurde.

Wird eine Variable/Konstante z. B. im Kopf einer Schleife vereinbart, gehört sie laut C99-Standard zu dem Block, in dem auch der Code der Schleife steht. Folgender Codeausschnitt soll das verdeutlichen:

```
for (int i = 0; i < 10; i++)
{
  printf("i: %d\n", i); // Ausgabe von lokal deklarierter Schleifenvariable
}
printf("i: %d\n", i); // Compilerfehler: hier ist i nicht mehr gültig!
```

Existiert in einem Block eine Variable mit einem Namen, der auch im umgebenden Block verwendet wird, so greift man im inneren Block über den Namen auf die Variable des inneren Blocks zu, die äußere wird *überdeckt*.

```
#include <stdio.h>

int main()
{
  int v = 1;
  int w = 5;
  {
    int v;
    v = 2;
    printf("%d\n", v);
    printf("%d\n", w);
  }
  printf("%d\n", v);
  return 0;
}
```

Nach der Kompilierung und Ausführung des Programms erhält man die folgende Ausgabe:

```
2
5
1
```

Erklärung: Am Anfang des neuen Anweisungsblocks in Zeile 8, wird eine neue Variable vdefiniert und ihr der Wert 2 zugewiesen. Die innere Variable v"überdeckt" nun den Wert der Variable vdes äußeren Blocks. Aus diesem Grund wird in Zeile 10 auch der Wert 2 ausgegeben. Nachdem der Gültigkeitsbereich der inneren Variable vin Zeile 12 verlassen wurde, existiert sie nicht mehr, so dass sie nicht mehr die äußere Variable überdecken kann. In Zeile 13 wird deshalb der Wert 1 ausgeben.

Sollte es in geschachtelten Anweisungblöcken nicht zu solchen Überschneidungen von Namen kommen, kann in einem inneren Block auf die Variablen des äußeren zugegriffen werden. In Zeile 11 kann deshalb die in Zeile 6 definierte Zahl wausgegeben werden.

12 Kapitel 7 auf Seite 65
13 Kapitel 6.2 auf Seite 54

en:C Programming/Variables[14] fr:Programmation C/Bases du langage[15] it:C/Variabili, operatori e costanti/Variabili[16] ja:C言語変数[17] pt:Programar em C/Variáveis[18]

14 http://en.wikibooks.org/wiki/C%20Programming%2FVariables
15 http://fr.wikibooks.org/wiki/Programmation%20C%2FBases%20du%20langage
16 http://it.wikibooks.org/wiki/C%2FVariabili%2C%20operatori%20e%20costanti%2FVariabili
17 http://ja.wikibooks.org/wiki/C%E8%A8%80%E8%AA%9E%20%E5%A4%89%E6%95%B0
18 http://pt.wikibooks.org/wiki/Programar%20em%20C%2FVari%C3%A1veis

3 static & Co.

Manchmal reichen einfache Variablen, wie sie im vergangenen Kapitel behandelt werden, nicht aus, um ein Problem zu lösen. Deshalb stellt der C-Standard einige Operatoren zur Verfügung, mit denen man das Verhalten einer Variablen weiter präzisieren kann.

3.1 static

Das Schlüsselwort `static`hat in C eine Doppelbedeutung. Im Kontext einer Variablendeklaration innerhalb einer Funktion sagt dieses Schlüsselwort, dass diese Variable auf einer festen Speicheradresse gespeichert wird. Daraus ergibt sich die Möglichkeit, dass eine Funktion, die mit `static`-Variablen arbeitet, beim nächsten Durchlauf die Informationen erneut nutzt, die in der Variablen gespeichert wurden (wie in einem Gedächtnis). Siehe dazu folgenden Code einer fiktiven Login-Funktion:

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

int login(const char user[], const char passwort[]) {
   static int versuch = 0;   /* erzeugen einer static-Variablen mit Anfangswert
 0 */

   if ( versuch < 3 )  {
       if ( checkuser(user) != checkpass(passwort) ) {
            versuch++;
       } else {
           versuch=0;
           return EXIT_SUCCESS;
       }
   }
   return EXIT_FAILURE;
}
```

Die in Zeile 5 erzeugte Variable zählt die Anzahl der Versuche und gibt nach 3 Fehlversuchen immer einen Fehler aus, auch wenn der Benutzer danach das richtige Passwort hätte. Wenn vor den 3 Versuchen das richtige Passwort gewählt wurde, wird der Zähler zurück gesetzt und man hat 3 neue Versuche. (Achtung! Dies ist nur ein Lehrbeispiel. Bitte niemals so einen Login realisieren, da diese Funktion z.B. nicht mit mehr als einem Benutzer arbeiten kann.)

In der Zeile 5 wird die Variable `versuch`mit dem Wert 0 initialisiert. Bei einer Variablen ohne das zusätzliche Attribut `static`hätte dies die Konsequenz, dass die Variable bei jedem Aufruf der Funktion `login`erneut initialisiert würde. Im obigen Beispiel könnte die Variable `versuch`damit niemals den Wert 3 erreichen. Das Programm wäre fehlerhaft. Statische Variablen werden hingegen nur einmal initialisiert, und zwar vom Compiler. Der Compiler

erzeugt eine ausführbare Datei, in der an der Speicherstelle für die statische Variable bereits der Initialisierungswert eingetragen ist.

Auch vor Funktionen sowie Variablen außerhalb von Funktionen kann das Schlüsselwort `static`stehen. Das bedeutet, dass auf die Funktion/Variable nur in der Datei, in der sie steht, zugegriffen werden kann.

```
static int checklogin(const char user[], const char passwort[]) {
   if( strcmp(user, "root") == 0 ) {
      if( strcmp(passwort, "default") == 0 ) {
         return 1;
      }
   }
   return 0;
}
```

Bei diesem Quelltext wäre die Funktion `checklogin`nur in der Datei sichtbar, in der sie definiert wurde.

3.2 volatile

Der Operator sagt dem Compiler, dass der Inhalt einer Variablen sich außerhalb des normalen Programmflusses ändern kann. Das kann zum Beispiel dann passieren, wenn ein Programm aus einer Interrupt-Service-Routine einen Wert erwartet und dann über diesen Wert einfach pollt (kein schönes Verhalten, aber gut zum Erklären von `volatile`). Siehe folgendes Beispiel

```
char keyPressed;
int count=0;

while (keyPressed != 'x') {
   count++;
}
```

Viele Compiler werden aus der `while`-Schleife ein `while(1)`machen, da sich der Wert von `keyPressed`aus ihrer Sicht nirgendwo ändert. Der Compiler könnte annehmen, dass der Ausdruck `keyPressed != 'x'`niemals unwahr werden kann. *Achtung:*Nur selten geben Compiler hier eine Warnung aus. Wenn Sie jetzt aber eine Systemfunktion geschrieben haben, die in die Adresse von `keyPressed`die jeweilige gedrückte Taste schreibt, kann das oben Geschriebene sinnvoll sein. In diesem Fall müssten Sie vor der Deklaration von `keyPressed`die Erweiterung `volatile`schreiben, damit der Compiler von seiner vermeintlichen Optimierung absieht. Siehe richtiges Beispiel:

```
volatile char keyPressed;
int count=0;

while (keyPressed != 'x') {
   count++;
}
```

Das Keyword `volatile`sollte sparsam verwendet werden, da es dem Compiler jegliches Optimieren verbietet.

3.3 register

Dieses Schlüsselwort ist ein Optimierungshinweis an den Compiler. Zweck von `register`ist es, dem Compiler mitzuteilen, dass man die so gekennzeichnete Variable häufig nutzt und dass es besser wäre, sie direkt in ein Register des Prozessors abzubilden. Normalerweise werden Variablen auf dem Stapel (engl. *stack*) abgelegt. Register können jedoch sehr viel schneller gelesen und geschrieben werden als der Stack. Der Compiler kann sehr effizienten Code generieren, wenn er weiß, dass eine Variable im Prozessor-Register gespeichert werden darf. Dies kann zum Beispiel bei Schleifenzählern und dergleichen sinnvoll sein.

Moderne Compiler sind meistens so intelligent, dass das Schlüsselwort `register`getrost weggelassen werden kann. In der Regel ist es viel besser, die Optimierung des Codes ganz dem Compiler zu überlassen.

Es sollte weiterhin beachtet werden, dass das Schlüsselwort `register`vom Compiler ignoriert werden kann. Er ist nicht gezwungen, eine so gekennzeichnete Variable in ein Prozessor-Register abzubilden.

Ein anderer Effekt des Schlüsselworts `register`ist, dass es nicht möglich ist, auf die Adresse von einer mit `register`qualifizierten Variablen zuzugreifen. Dies gilt unabhängig davon, ob die Variable tatsächlich in einem Register platziert wurde oder nicht. Selbst wenn der Compiler die `register`Anweisung in dem Sinne ignoriert, dass die Variable nicht in ein register platziert wird, so kann der Programmierer trotzdem nicht mittels eines Pointers auf die Adresse der Variablen zugreifen.

4 Einfache Ein- und Ausgabe

Wohl kein Programm kommt ohne Ein- und Ausgabe aus. In C ist die Ein-/Ausgabe allerdings kein Bestandteil der Sprache selbst. Vielmehr liegen Ein- und Ausgabe als eigenständige Funktionen vor, die dann durch den Linker eingebunden werden. Die wichtigsten Ein- und Ausgabefunktionen werden Sie in diesem Kapitel kennenlernen.

4.1 printf

Die Funktion `printf`haben wir bereits in unseren bisherigen Programmen benutzt. Zeit also, sie genauer unter die Lupe zu nehmen. Die Funktion `printf`hat die folgende Syntax:

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

Bevor wir aber `printf`diskutieren, sehen wir uns noch einige Grundbegriffe von Funktionen an. In einem späteren Kapitel[1] werden Sie dann lernen, wie Sie eine Funktion selbst schreiben können.

In den beiden runden Klammern befinden sich die *Parameter*. In unserem Beispiel ist der Parameter `const char *format`. Die drei Punkte dahinter zeigen an, dass die Funktion noch weitere Parameter erhalten kann. Die Werte, die der Funktion übergeben werden, bezeichnet man als *Argumente*. In unserem „Hallo Welt"-Programm haben wir der Funktion `printf`beispielsweise das Argument "Hallo Welt" übergeben.

Außerdem kann eine Funktion einen *Rückgabewert*besitzen. In diesem Fall ist der Typ des Rückgabewertes `int`. Den Typ der Rückgabe erkennt man am Schlüsselwort, das vor der Funktion steht. Eine Funktion, die keinen Wert zurückgibt, erkennen Sie an dem Schlüsselwort `void`.

Die Bibliotheksfunktion `printf`dient dazu, eine Zeichenkette (engl. String) auf der Standardausgabe auszugeben. In der Regel ist die Standardausgabe der Bildschirm. Als Übergabeparameter besitzt die Funktion einen Zeiger auf einen konstanten String. Was es mit Zeigern auf sich hat, werden wir später[2] noch sehen. Das `const`bedeutet hier, dass die Funktion den String nicht verändert. Über den Rückgabewert liefert `printf`die Anzahl der ausgegebenen Zeichen. Wenn bei der Ausgabe ein Fehler aufgetreten ist, wird ein negativer Wert zurückgegeben.

Als erstes Argument von `printf`sind nur Strings erlaubt. Bei folgender Zeile gibt der Compiler beim Übersetzen deshalb eine Warnung oder einen Fehler aus:

---

1 Kapitel 7 auf Seite 65
2 Kapitel 9 auf Seite 77

```
printf(55); // falsch
```

Da die Anführungszeichen fehlen, nimmt der Compiler an, dass es sich bei 55 um einen Zahlenwert handelt. Geben Sie dagegen 55 in Anführungszeichen an, interpretiert der Compiler dies als Text. Bei der folgenden Zeile gibt der Compiler deshalb keinen Fehler aus:

```
printf("55"); // richtig
```

4.1.1 Formatelemente von printf

Die `printf`-Funktion kann auch mehrere Parameter verarbeiten, diese müssen dann durch *Kommata*voneinander getrennt werden.

Beispiel:

```
#include <stdio.h>

int main()
{
  printf("%i plus %i ist gleich %s.", 3, 2, "Fünf");
  return 0;
}
```

Ausgabe:

```
3 plus 2 ist gleich Fünf.
```

Die mit dem %-Zeichen eingeleiteten Formatelemente greifen nacheinander auf die durch Komma getrennten Parameter zu (das erste `%i`auf 3, das zweite `%i`auf 2 und `%s`auf den String ”Fünf”).

Innerhalb von `format`werden *Umwandlungszeichen*(engl. conversion modifier) für die weiteren Parameter eingesetzt. Hierbei muss der richtige Typ verwendet werden. Die wichtigsten Umwandlungszeichen sind:

Zeichen	**Umwandlung**
%d oder %i	int
%c	einzelnes Zeichen
%e oder %E	double im Format [-]d.ddd e±dd bzw. [-]d.ddd E±dd
%f	double im Format [-]ddd.ddd
%o	int als Oktalzahl ausgeben
%p	die Adresse eines Pointers[3]
%s	Zeichenkette ausgeben
%u	unsigned int
%x oder %X	int als Hexadezimalzahl ausgeben
%%	Prozentzeichen

3 Kapitel 9 auf Seite 77

Weitere Formate und genauere Erläuterungen finden Sie in der Referenz[4] dieses Buches.

Beispiel:

```
#include <stdio.h>

int main()
{
  printf("Integer: %d\n", 42);
  printf("Double: %.6f\n", 3.141);
  printf("Zeichen: %c\n", 'z');
  printf("Zeichenkette: %s\n", "abc");
  printf("43 Dezimal ist in Oktal: %o\n", 43);
  printf("43 Dezimal ist in Hexadezimal: %x\n", 43);
  printf("Und zum Schluss geben wir noch das Prozentzeichen aus: %%\n");

  return 0;
}
```

Nachdem Sie das Programm übersetzt und ausgeführt haben, erhalten Sie die folgende Ausgabe:

```
Integer: 42
Double: 3.141000
Zeichen: z
Zeichenkette: abc
43 Dezimal ist in Oktal: 53
43 Dezimal ist in Hexadezimal: 2b
Und zum Schluss geben wir noch das Prozentzeichen aus: %
```

Neben dem Umwandlungszeichen kann eine Umwandlungsangabe weitere Elemente zur Formatierung erhalten. Dies sind maximal:

- ein **Flag**
- die **Feldbreite**
- durch einen Punkt getrennt die Anzahl der **Nachkommstellen**(Längenangabe)
- und an letzter Stelle schließlich das Umwandlungszeichen selbst

4.1.2 Flags

Unmittelbar nach dem Prozentzeichen werden die *Flags*(dt. Kennzeichnung) angegeben. Sie haben die folgende Bedeutung:

- `-`(Minus): Der Text wird links ausgerichtet.
- `+`(Plus): Es wird auch bei einem positiven Wert ein Vorzeichen ausgegeben.
- Leerzeichen: Ein Leerzeichen wird ausgegeben, wenn der Wert positiv ist.
- `#`: Welche Wirkung das Kennzeichen `#`hat, ist abhängig vom verwendeten Format: Wenn ein Wert über `%x`als Hexadezimal ausgegeben wird, so wird jedem Wert ein 0x vorangestellt (außer der Wert ist 0).
- `0`: Die Auffüllung erfolgt mit Nullen anstelle von Leerzeichen, wenn die Feldbreite verändert wird.

4 Kapitel 22.10.2 auf Seite 177

Im folgenden ein Beispiel, das die Anwendung der Flags zeigt:

```
#include <stdio.h>

int main()
{
   printf("Zahl 67:%+i\n", 67);
   printf("Zahl 67:% i\n", 67);
   printf("Zahl 67:%#x\n", 67);
   printf("Zahl 0:%0x\n", 0);
   return 0;
}
```

Wenn das Programm übersetzt und ausgeführt wird, erhalten wir die folgende Ausgabe:

```
Zahl 67:+67
Zahl 67: 67
Zahl 67:0x43
Zahl 0:0
```

4.1.3 Feldbreite

Hinter dem Flag kann die *Feldbreite*(engl. field width) festgelegt werden. Das bedeutet, dass die Ausgabe mit der entsprechenden Anzahl von Zeichen aufgefüllt wird. Beispiel:

```
int main()
{
  printf("Zahlen rechtsbündig ausgeben: %5d, %5d, %5d\n",34, 343, 3343);
  printf("Zahlen rechtsbündig ausgeben, links mit 0 aufgefüllt: %05d, %05d,
 %05d\n",34, 343, 3343);
  printf("Zahlen linksbündig ausgeben: %-5d, %-5d, %-5d\n",34, 343, 3343);
  return 0;
}
```

Wenn das Programm übersetzt und ausgeführt wird, erhalten wir die folgende Ausgabe:

```
Zahlen rechtsbündig ausgeben:   34,   343,  3343
Zahlen rechtsbündig ausgeben, links mit 0 aufgefüllt: 00034, 00343, 03343
Zahlen linksbündig ausgeben: 34   , 343  , 3343
```

In Zeile 4 haben wir anstelle der Leerzeichen eine 0 verwendet, so dass nun die Feldbreite mit Nullen aufgefüllt wird.

Standardmäßig erfolgt die Ausgabe rechtsbündig. Durch Voranstellen des Minuszeichens kann die Ausgabe aber auch linksbündig erfolgen, wie in Zeile 5 zu sehen ist.

4.1.4 Nachkommastellen

Nach der Feldbreite folgt, durch einen Punkt getrennt, die *Genauigkeit*. Bei `%f`werden ansonsten standardmäßig 6 Nachkommastellen ausgegeben. Diese Angaben sind natürlich auch nur bei den Gleitkommatypen `float`und `double`sinnvoll, weil alle anderen Typen keine Nachkommastellen besitzen.

Beispiel:

```
#include <stdio.h>

int main()
{
  double betrag1 = 0.5634323;
  double betrag2 = 0.2432422;
  printf("Summe: %.3f\n", betrag1 + betrag2);

  return 0;
}
```

Wenn das Programm übersetzt und ausgeführt wurde, erscheint die folgende Ausgabe auf dem Bildschirm:

```
  Summe: 0.807
```

4.2 scanf

Auch die Funktion `scanf`haben Sie bereits kennengelernt. Sie hat eine vergleichbare Syntax wie `printf`:

```
int scanf (const char *format, ...);
```

Die Funktion `scanf`liest einen Wert ein und speichert diesen in den angegebenen Variablen ab. Doch Vorsicht: Die Funktion `scanf`erwartet die Adresse der Variablen. Deshalb führt der folgende Funktionsaufruf zu einem **Fehler**:

```
scanf("%i", x); /* Fehler */
```

Richtig dagegen ist:

```
scanf("%i",&x);
```

Mit dem *Adressoperator*& erhält man die Adresse einer Variablen. Diese kann man sich auch ausgeben lassen:

```
#include <stdio.h>

int main(void)
{
  int x = 5;

  printf("Adresse von x: %p\n", &x);
  printf("Inhalt der Speicherzelle: %d\n", x);

   return 0;
}
```

Kompiliert man das Programm und führt es aus, erhält man z.B. die folgende Ausgabe:

```
Adresse von x: 0022FF74
Inhalt der Speicherzelle: 5
```

Die Ausgabe der Adresse kann bei Ihnen variieren. Es ist sogar möglich, dass sich diese Angabe bei jedem Neustart des Programms ändert. Dies hängt davon ab, wo das Programm (vom Betriebssystem) in den Speicher geladen wird.

Mit Adressen werden wir uns im Kapitel Zeiger[5] noch einmal näher beschäftigen.

Für `scanf`können die folgenden Platzhalter verwendet werden, die dafür sorgen, dass der eingegebene Wert in das "richtige" Format umgewandelt wird:

Zeichen	**Umwandlung**
%d	vorzeichenbehafteter Integer als Dezimalwert
%i	vorzeichenbehafteter Integer als Dezimal-, <br \>Hexadezimal oder Oktalwert
%e, %f, %g	Fließkommazahl
%o	int als Oktalzahl einlesen
%s	Zeichenkette einlesen
%x	Hexadezimalwert
%%	erkennt das Prozentzeichen

4.3 getchar und putchar

Die Funktion `getchar`liefert das nächste Zeichen vom Standard-Eingabestrom. Ein *Strom*(engl. stream) ist eine geordnete Folge von Zeichen, die als Ziel oder Quelle ein Gerät hat. Im Falle von `getchar`ist dieses Gerät die Standardeingabe -- in der Regel also die Tastatur. Der Strom kann aber auch andere Quellen oder Ziele haben: Wenn wir uns später noch mit dem Speichern und Laden von Dateien[6] beschäftigen, dann ist das Ziel und die Quelle des Stroms eine Datei.

Das folgende Beispiel liest ein Zeichen von der Standardeingabe und gibt es aus. Eventuell müssen Sie nach der Eingabe des Zeichens <Enter> drücken, damit überhaupt etwas passiert. Das liegt daran, dass die Standardeingabe üblicherweise zeilenweise und nicht zeichenweise eingelesen wird.

```
int c;
c = getchar();
putchar(c);
```

Geben wir über die Tastatur "hallo" ein, so erhalten wir durch den Aufruf von `getchar`zunächst das erste Zeichen (also das "h"). Durch einen erneuten Aufruf von `getchar`erhalten wir das nächste Zeichen, usw. Die Funktion `putchar(c)`ist quasi die Umkehrung von `getchar`: Sie gibt ein einzelnes Zeichen *c*auf der Standardausgabe aus. In der Regel ist die Standardausgabe der Monitor.

5 Kapitel 9 auf Seite 77
6 Kapitel 18 auf Seite 139

Zugegeben, die Benutzung von `getchar`hat hier wenig Sinn, außer man hat vor, nur das erste Zeichen einer Eingabe einzulesen. Häufig wird `getchar`mit Schleifen[7] benutzt. Ein Beispiel dafür werden wir noch später[8] kennenlernen.

4.4 Escape-Sequenzen

Eine spezielle Darstellung kommt in C den Steuerzeichen[9] zugute. Steuerzeichen sind Zeichen, die nicht direkt auf dem Bildschirm sichtbar werden, sondern eine bestimmte Aufgabe erfüllen, wie etwa das Beginnen einer neuen Zeile, das Darstellen des Tabulatorzeichens oder das Ausgeben eines Warnsignals. So führt beispielsweise

```
printf("Dies ist ein Text ");
printf("ohne Zeilenumbruch");
```

nicht etwa zu dem Ergebnis, dass nach dem Wort „Text“ eine neue Zeile begonnen wird, sondern das Programm gibt nach der Kompilierung aus:

```
Dies ist ein Text ohne Zeilenumbruch
```

Eine neue Zeile wird also nur begonnen, wenn an der entsprechenden Stelle ein `\n`steht. Die folgende Auflistung zeigt alle in C vorhandenen Escape-Sequenzen[10]:

- `\n`(new line) = bewegt den Cursor auf die Anfangsposition der nächsten Zeile.
- `\t`(horizontal tab) = Setzt den Tabulator auf die nächste horizontale Tabulatorposition. Wenn der Cursor bereits die letzte Tabulatorposition erreicht hat, dann ist das Verhalten unspezifiziert (vorausgesetzt eine letzte Tabulatorposition existiert).
- `\a`(alert) = gibt einen hör- oder sichtbaren Alarm aus, ohne die Position des Cursors zu ändern
- `\b`(backspace) = Setzt den Cursor ein Zeichen zurück. Wenn sich der Cursor bereits am Zeilenanfang befindet, dann ist das Verhalten unspezifiziert.
- `\r`(carriage return, dt. Wagenrücklauf) = Setzt den Cursor an den Zeilenanfang
- `\f`(form feed) = Setzt den Cursor auf die Startposition der nächsten Seite.
- `\v`(vertical tab) = Setzt den Cursor auf die nächste vertikale Tabulatorposition. Wenn der Cursor bereits die letzte Tabulatorposition erreicht hat, dann ist das Verhalten unspezifiziert (wenn eine solche existiert).
- `\"`” wird ausgegeben
- `\'`’ wird ausgegeben
- `\?`? wird ausgegeben
- `\\`\ wird ausgegeben
- `\0`ist die Endmarkierung einer Zeichenkette

Jede Escape-Sequenz symbolisiert ein Zeichen auf einer Implementierung und kann in einer Variablen des Typs `char`gespeichert werden.

7 Kapitel 6.2 auf Seite 54
8 Kapitel 6.2.2 auf Seite 56
9 `http://de.wikipedia.org/wiki/Steuerzeichen`
10 `http://de.wikipedia.org/wiki/Escape-Sequenzen`

Beispiel:

```
#include <stdio.h>

int main(void)
{
  printf("Der Zeilenumbruch erfolgt\n");
  printf("durch die Escape-Sequenz \\n\n\n");
  printf("Im Folgenden wird ein Wagenrücklauf (carriage return) mit \\r
 erzeugt:\r");
  printf("Satzanfang\n\n");
  printf("Folgende Ausgabe demonstriert die Funktion von \\b\n");
  printf("12\b34\b56\b78\b9\n");
  printf("Dies ist lesbar\n\0und dies nicht mehr.");      /* erzeugt ggf. eine
 Compiler-Warnung */
  return 0;
}
```

Erzeugt auf dem Bildschirm folgende Ausgabe:

```
Der Zeilenumbruch erfolgt
durch die Escape-Sequenz \n

Satzanfangen wird ein Wagenrücklauf (carriage return) mit \r erzeugt:

Folgende Ausgabe demonstriert die Funktion von \b
13579
Dies ist lesbar
```

5 Operatoren

5.1 Grundbegriffe

Bevor wir uns mit den Operatoren näher beschäftigen, wollen wir uns noch einige Grundbegriffe ansehen:

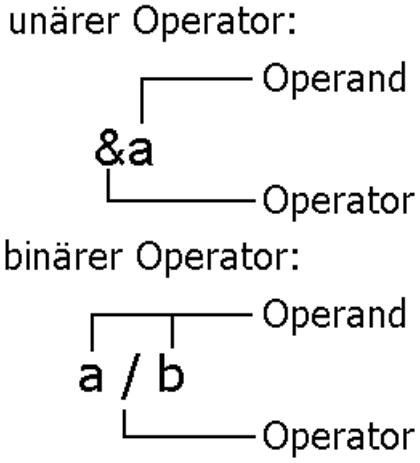


Abb. 1 Unäre und binäre Operatoren

Man unterscheidet in der Sprache C *unäre*, *binäre*und *ternäre*Operatoren. Unäre Operatoren besitzen nur einen Operanden, binäre Operatoren besitzen zwei Operanden und ternäre drei. Ein unärer Operator ist beispielsweise der `&`-Operator, ein binärer Operator der Geteilt-Operator (`/`). Es gibt auch Operatoren, die, je nachdem wo sie stehen, entweder unär oder binär sind. Ein Beispiel hierfür sind Plus (`+`) und Minus (`-`). Sie können als Vorzeichen vorkommen und sind dann unäre Operatoren oder als Rechenzeichen und sind dann binäre Operatoren. Der einzige ternäre Operator in C ist der Bedingungsoperator, der weiter unten behandelt wird.

Sehr häufig kommen im Zusammenhang mit binären Operatoren auch die Begriffe **L- und R-Wert**vor. Diese Begriffe stammen ursprünglich von Zuweisungen. Der Operand links des Zuweisungsoperators wird als L-Wert (engl. L value) bezeichnet, der Operand rechts als R-Wert (engl. R value). Verallgemeinert gesprochen sind L-Werte Operanden, denen man einen Wert zuweisen kann, R-Werten kann kein Wert zugewiesen werden. Alle beschreibbaren Variablen sind also L-Werte. Konstanten, Literale und konstante Zeichenketten (String Literalen) hingegen sind R-Werte. Je nach Operator dürfen bestimmte Operanden nur L-Werte sein. Beim Zuweisungsoperator muss beispielsweise der erste Operand ein L-Wert sein.

```
a = 35;
```

In der Zuweisung ist der erste Operand die Variable a (ein L-Wert), der zweite Operand das Literal 35 (ein R-Wert). Nicht erlaubt hingegen ist die Zuweisung

```
35 = a; /* Fehler */
```

da einem Literal kein Wert zugewiesen werden darf. Anders ausgedrückt: Ein Literal ist kein L-Wert und darf deshalb beim Zuweisungsoperator nicht als erster Operand verwendet werden. Auch bei anderen Operatoren sind nur L-Werte als Operand erlaubt. Ein Beispiel hierfür ist der Adressoperator. So ist beispielsweise auch der folgende Ausdruck falsch:

```
&35; /* Fehler */
```

Der Compiler wird eine Fehlermeldung ausgeben, in welcher er vermutlich darauf hinweisen wird, dass hinter dem `&`-Operator ein L-Wert folgen muss.

5.2 Inkrement- und Dekrement-Operator

Mit den `++`- und `---`Operatoren kann ein L-Wert um eins erhöht bzw. um eins vermindert werden. Man bezeichnet die Erhöhung um eins auch als *Inkrement*, die Verminderung um eins als *Dekrement*. Ein Inkrement einer Variable `x`entspricht `x = x + 1`, ein Dekrement einer Variable `x`entspricht `x = x - 1`.

Der Operator kann sowohl vor als auch nach dem Operanden stehen. Steht der Operator vor dem Operand, spricht man von einem *Präfix*, steht er hinter dem Operand bezeichnet man ihn als *Postfix*. Je nach Kontext unterscheiden sich die beiden Varianten, wie das folgende Beispiel zeigt:

```
x = 10;
 ergebnis = ++x;
```

Die zweite Zeile kann gelesen werden als: ”Erhöhe zunächst `x`um eins, und weise dann den Wert der Variablen zu”. Nach der Zuweisung besitzt sowohl die Variable `ergebnis`wie auch die Variable `x`den Wert 11.

```
x = 10;
 ergebnis = x++;
```

Die zweite Zeile kann nun gelesen werden als: ”Weise der Variablen `ergebnis`den Wert`x`zu und erhöhe anschließend `x`um eins.” Nach der Zuweisung hat die Variable `ergebnis`deshalb den Wert 10, die Variable `x`den Wert 11.

Der `++`- bzw. `---`Operator sollte, wann immer es möglich ist, präfix verwendet werden, da schlechte und ältere Compiler den Wert des Ausdruckes sonst (unnötigerweise) zuerst kopieren, dann erhöhen und dann in die Variable zurückschreiben. So wird aus

```
i++
```

schnell

```
int j = i;
j = j + 1;
i = j;
```

wobei der Mehraufwand hier deutlich ersichtlich ist. Auch wenn man später zu C++[1] wechseln will, sollte man sich von Anfang an den Präfixoperator angewöhnen, da die beiden Anwendungsweisen dort fundamental anders sein können.

5.3 Rangfolge und Assoziativität

Wie Sie bereits im ersten Kapitel gesehen haben, besitzen der Mal- und der Geteilt-Operator eine höhere **Rangfolge (auch als Priorität bezeichnet)**als der Plus- und der Minus-Operator. Diese Regel ist Ihnen sicher noch aus der Schule als "Punkt vor Strich" bekannt.

Was ist mit einem Ausdruck wie beispielsweise:

```
c = sizeof(x) + ++a / 3;
```

In C hat jeder Operator eine Rangfolge, nach der der Compiler einen Ausdruck auswertet. Diese Rangfolge finden Sie in der Referenz[2] dieses Buches.

Der `sizeof()`- sowie der Präfix-Operator haben die Priorität 14, `+`die Priorität 12 und `/`die Priorität 13 [3].

Folglich wird der Ausdruck wie folgt ausgewertet:

```
c = (sizeof(x)) + ((++a) / 3);
```

Neben der Priorität ist bei Operatoren mit der gleichen Priorität auch die **Reihenfolge (auch als Assoziativität bezeichnet)**der Auswertung von Bedeutung. So muss beispielsweise der Ausdruck

```
4 / 2 / 2
```

von links nach rechts ausgewertet werden:

```
(4 / 2) / 2   // ergibt 1
```

Wird die Reihenfolge dieser Auswertung geändert, so ist das Ergebnis falsch:

```
4 / (2 / 2)   // ergibt 4
```

In diesem Beispiel ist die Auswertungsreihenfolge

1 `http://de.wikibooks.org/wiki/C%2B%2B-Programmierung`

2 Kapitel 22.6.10 auf Seite 172

3 Die Rangfolge der Operatoren ist im Standard nicht in Form einer Tabelle festgelegt, sondern ergibt sich aus der Grammatik ^{`http://de.wikipedia.org/wiki/Formale%20Grammatik%20`} der Sprache C. Deshalb können sich die Werte für die Rangfolge in anderen Büchern unterscheiden, wenn eine andere Zählweise verwendet wurde, andere Bücher verzichten wiederum vollständig auf die Durchnummerierung der Rangfolge.

```
(4 / 2) / 2
```

, also **linksassoziativ**.

Nicht alle Ausdrücke werden aber von links nach rechts ausgewertet, wie das folgende Beispiel zeigt:

```
a = b = c = d;
```

Durch Klammerschreibweise verdeutlicht, wird dieser Ausdruck vom Compiler von rechts nach links ausgewertet:

```
a = (b = (c = d));
```

Der Ausdruck ist also **rechtsassoziativ**.

Dagegen lässt sich auf das folgende Beispiel die Assoziativitätsregel nicht anwenden:

```
5 + 4 * 8 + 2
```

Sicher sieht man bei diesem Beispiel sofort, dass es wegen "Punkt vor Strich" keinen Sinn macht, eine bestimmte Bewertungsreihenfolge festzulegen. Uns interessiert hier allerdings die Begründung die C hierfür liefert: Diese besagt, wie wir bereits wissen, dass die Assoziativitätsregel nur auf Operatoren mit gleicher Priorität anwendbar ist. Der Plusoperator hat allerdings eine geringere Priorität als der Multiplikationsoperator.

Diese Assoziativität von jedem Operator finden Sie in der Referenz[4] dieses Buches.

Durch unsere bisherigen Beispiele könnte der Anschein erweckt werden, dass alle Ausdrücke ein definiertes Ergebnis besitzen. Leider ist dies nicht der Fall.

Fast alle C-Programme besitzen sogenannte **Nebenwirkungen**(engl. side effect; teilweise auch mit Seiteneffekt übersetzt). Als Nebenwirkungen bezeichnet man die Veränderung des Zustandes des Rechnersystems durch das Programm. Typische Beispiele hierfür sind Ausgabe, Eingabe und die Veränderung von Variablen. Beispielsweise führt `i++`zu einer Nebenwirkung - die Variable wird um eins erhöht.

Der C-Standard legt im Programm bestimmte Punkte fest, bis zu denen Nebenwirkungen ausgewertet sein müssen. Solche Punkte werden als *Sequenzpunkte*(engl. sequence point) bezeichnet. In welcher Reihenfolge die Nebenwirkungen vor dem Sequenzpunkt auftreten und welche Auswirkungen dies hat, ist nicht definiert.

Die folgenden Beispiele sollten dies verdeutlichen:

```
i = 3;
a = i + i++;
```

Da der zweite Operand der Addition ein Postfix-Inkrement-Operator ist, wird dieser zu 3 ausgewertet. Je nachdem, ob der erste Operand vor oder nach Einsetzen der Nebenwirkung ausgewertet wird (also ob `i`noch 3 oder schon 4 ist), ergibt die Addition 6 oder 7. Da sich der Sequenzpunkt aber am Ende der Zeile befindet, ist beides möglich und C-konform. Um

4 Kapitel 22.6.10 auf Seite 172

es nochmals hervorzuheben: Nach dem Sequenzpunkt besitzt `i`in jedem Fall den Wert 4. Es ist allerdings nicht definiert, wann `i`inkrementiert wird. Dies kann vor oder nach der Addition geschehen.

Ein weiterer Sequenzpunkt befindet sich vor dem Eintritt in eine Funktion. Hierzu zwei Beispiele:

```
a = 5;
printf("Ausgabe: %d %d",a += 5,a *= 2);
```

Die Ausgabe kann entweder 10 20 oder 15 10 sein, je nachdem ob die Nebenwirkung von `a += 5`oder `a*= 2`zuerst ausgeführt wird.

Zweites Beispiel:

```
x = a() + b() - c();
```

Wie wir oben gesehen haben, ist festgelegt, dass der Ausdruck von links nach rechts ausgewertet wird (`(a() + b()) - c()`), da der Ausdruck linksassoziativ ist. Allerdings steht damit nicht fest, welche der Funktionen als erstes aufgerufen wird. Der Aufruf kann in den Kombinationen a, b, c oder a, c, b oder b, a, c oder b, c, a oder c, a, b oder c, b, a erfolgen. Welche Auswirkungen dies auf die Programmausführung hat, ist undefiniert.

Weitere wichtige Sequenzpunkte sind die Operatoren `&&`, `||`sowie `?:`und Komma. Auf die Bedeutung dieser Operatoren werden wir noch im nächsten Kapitel[5] näher eingehen.

Es sei nochmals darauf hingewiesen, dass dies nicht wie im Fall eines implementierungsabhängigen oder unspezifizierten Verhalten zu Programmen führt, die nicht portabel sind. Vielmehr sollten Programme erst gar kein undefiniertes Verhalten liefern. Fast alle Compiler geben hierbei keine Warnung aus. Ein undefiniertes Verhalten kann allerdings buchstäblich zu allem führen. So ist es genauso gut möglich, dass der Compiler ein ganz anderes Ergebnis liefert als das Oben beschriebene, oder sogar zu anderen unvorhergesehenen Ereignissen wie beispielsweise dem Absturz des Programms.

5.4 Der Shift-Operator

Die Operatoren `<<`und `>>`dienen dazu, den Inhalt einer Variablen bitweise um 1 nach links bzw. um 1 nach rechts zu verschieben (siehe Abbildung 1).

Beispiel:

```
#include <stdio.h>

int main()
{
  unsigned short int a = 350;
  printf("%u\n", a << 1);

  return 0;
}
```

5 Kapitel 6 auf Seite 49

Nach dem Kompilieren und Übersetzen wird beim Ausführen des Programms die Zahl 700 ausgegeben. Die Zahl hinter dem Leftshiftoperator << gibt an, um wie viele Bitstellen die Variable verschoben werden soll (in diesem Beispiel wird die Zahl nur ein einziges Mal nach links verschoben).

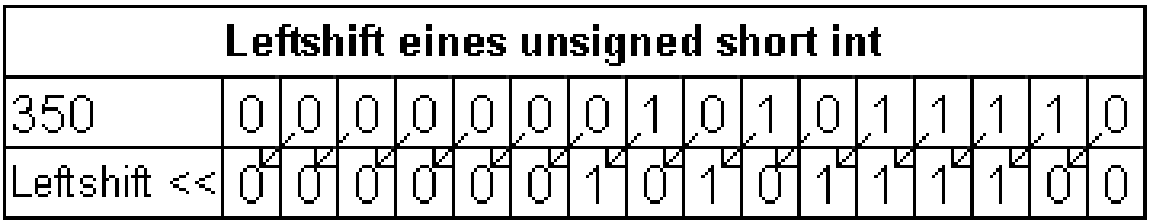


Abb. 2 Abb 1 Linksshift

Vielleicht fragen Sie sich jetzt, für was der Shift–Operator gut sein soll? Schauen Sie sich das Ergebnis nochmals genau an. Fällt Ihnen etwas auf? Richtig! Bei jedem Linksshift findet eine Multiplikation mit 2 statt. Umgekehrt findet beim Rechtsshift eine Division durch 2 statt. (Dies natürlich nur unter der Bedingung, dass die 1 nicht herausgeschoben wird und die Zahl positiv ist. Wenn der zu verschiebende Wert negativ ist, ist das Ergebnis implementierungsabhängig.)

Es stellt sich nun noch die Frage, weshalb man den Shift-Operator benutzen soll, wenn eine Multiplikation mit zwei doch ebenso gut mit dem *-Operator machbar wäre? Die Antwort lautet: Bei den meisten Prozessoren wird die Verschiebung der Bits wesentlich schneller ausgeführt als eine Multiplikation. Deshalb kann es bei laufzeitkritischen Anwendungen vorteilhaft sein, den Shift-Operator anstelle der Multiplikation zu verwenden. Eine weitere praktische Einsatzmöglichkeit des Shift Operators findet sich zudem in der Programmierung von Mikroprozessoren. Durch einen Leftshift können digitale Eingänge einfacher und schneller geschaltet werden. Man erspart sich hierbei mehrere Taktzyklen des Prozessors.

Anmerkung: Heutige Compiler optimieren dies schon selbst. Der Lesbarkeit halber sollte man also besser `x * 2`schreiben, wenn eine Multiplikation durchgeführt werden soll. Will man ein Byte als Bitmaske verwenden, d.h. wenn die einzelnen gesetzten Bits interessieren, dann sollte man mit Shift arbeiten, um seine Absicht im Code besser auszudrücken.

5.5 Ein wenig Logik ...

Kern der Logik sind Aussagen. Solche Aussagen sind beispielsweise:

- Stuttgart liegt in Baden-Württemberg.
- Der Himmel ist grün.
- 6 durch 3 ist 2.
- Felipe Massa wird in der nächsten Saison Weltmeister.

Aussagen können wahr oder falsch sein. Die erste Aussage ist wahr, die zweite dagegen falsch, die dritte Aussage dagegen ist wiederum wahr. Auch die letzte Aussage ist wahr oder falsch − allerdings wissen wir dies zum jetzigen Zeitpunkt noch nicht. In der Logik werden wahre Aussagen mit einer 1, falsche Aussagen mit einer 0 belegt. Was aber hat dies mit C zu tun? Uns interessieren hier Ausdrücke wie:

- 5 < 2 (fünf ist kleiner als zwei)
- 4 == 4 (gleich)
- 5 >= 2 (wird gelesen als: fünf ist größer oder gleich zwei)
- x > y (x ist größer als y)

Auch diese Ausdrücke können wahr oder falsch sein. Mit solchen sehr einfachen Ausdrücken kann der Programmfluss gesteuert werden. So kann der Programmierer festlegen, dass bestimmte Anweisungen nur dann ausgeführt werden, wenn beispielsweise x > y ist oder ein Programmabschnitt so lange ausgeführt wird wie a != b ist (in C bedeutet das Zeichen != immer ungleich).

Beispiel: Die Variable x hat den Wert 5 und die Variable y den Wert 7. Dann ist der Ausdruck x < y wahr und liefert eine 1 zurück. Der Ausdruck x > y dagegen ist falsch und liefert deshalb eine 0 zurück.

Für den Vergleich zweier Werte kennt C die folgenden Vergleichsoperatoren:

Operator	**Bedeutung**
<	kleiner als
>	größer als
<=	kleiner oder gleich
>=	größer oder gleich
!=	ungleich
==	gleich

Wichtig: Verwechseln Sie nicht den Zuweisungsoperator =mit dem Vergleichsoperator ==. Diese haben vollkommen verschiedene Bedeutungen. Während der erste Operator einer Variablen einen Wert zuweist, vergleicht letzterer zwei Werte miteinander. Da die Verwechslung der beiden Operatoren allerdings ebenfalls einen gültigen Ausdruck liefert, gibt der Compiler weder eine Fehlermeldung noch eine Warnung zurück. Dies macht es schwierig, den Fehler aufzufinden. Aus diesem Grund schreiben viele Programmierer grundsätzlich bei Vergleichen die Variablen auf die rechte Seite, also zum Beispiel 5 == a. Vergißt man mal ein =, wird der Compiler eine Fehlermeldung liefern.

Anders als in der Logik wird in C der boolsche Wert [6] true als Werte ungleich 0 definiert. Dies schließt auch beispielsweise die Zahl 5 ein, die in C ebenfalls als true interpretiert wird. Die Ursache hierfür ist, dass es in der ursprünglichen Sprachdefinition keinen Datentyp zur Darstellung der boolschen Werte true und false gab, so dass andere Datentypen zur Speicherung von boolschen Werten benutzt werden mussten. So schreibt beispielsweise der C-Standard vor, dass die Vergleichsoperatoren einen Wert vom Typ `int`liefern. Erst mit dem C99-Standard wurde ein neuer Datentyp `_Bool`eingeführt, der nur die Werte 0 und 1 aufnehmen kann.

6 Der Begriff boolsche Werte ist nach dem englischen Mathematiker George Boole ^{`http://de.wikipedia.org/wiki/George%20Boole%20`} benannt, der sich mit algebraischen Strukturen beschäftigte, die nur die Zustände 0 und 1 bzw. false und true kennt.

5.6 ... und noch etwas Logik

Wir betrachten die folgende Aussage:

Wenn ich morgen vor sechs Uhr Feierabend habe **und**das Wetter schön ist, dann gehe ich an den Strand.

Auch dies ist eine Aussage, die wahr oder die falsch sein kann. Im Unterschied zu den Beispielen aus dem vorhergegangen Kapitel, hängt die Aussage "gehe ich an den Strand" von den beiden vorhergehenden ab. Gehen wir die verschiedenen möglichen Fälle durch:

- Wir stellen am nächsten Tag fest, dass die Aussage, dass wir vor sechs Feierabend haben und dass das Wetter schön ist, falsch ist, dann ist auch die Aussage, dass wir an den Strand gehen, falsch.
- Wir stellen am nächsten Tag fest, die Aussage, dass wir vor sechs Feierabend haben, ist falsch, und die Aussage, dass das Wetter schön ist, ist wahr. Dennoch bleibt die Aussage, dass wir an den Strand gehen, falsch.
- Wir stellten nun fest, dass wir vor sechs Uhr Feierabend haben, also die Aussage wahr ist, aber dass die Aussage, dass das Wetter schön ist falsch ist. Auch in diesem Fall ist die Aussage, dass wir an den Strand gehen, falsch.
- Nun stellen wir fest, dass sowohl die Aussage, dass wir vor sechs Uhr Feierabend haben wie auch die Aussage, dass das Wetter schön ist wahr sind. In diesem Fall ist auch die Aussage, dass das wir an den Strand gehen, wahr.

Dies halten wir nun in einer Tabelle fest:

Eingabe 1	**Eingabe 2**	**Ergebnis**
falsch	falsch	falsch
falsch	wahr	falsch
wahr	falsch	falsch
wahr	wahr	wahr

In der Informatik nennt man dies eine Wahrheitstabelle -- in diesem Fall der UND- bzw. AND-Verknüpfung.

Eine UND-Verknüpfung in C wird durch den `&`-Operator repräsentiert. Beispiel:

```
int a;
a = 45 & 35
```

Bitte berücksichtigen Sie, dass bei boolschen Operatoren beide Operanden vom Typ Integer sein müssen.

Eine weitere Verknüpfung ist die Oder-Verknüpfung. Auch diese wollen wir uns an einem Beispiel klar machen:

Wenn wir eine Pizzeria **oder**ein griechisches Lokal finden, kehren wir ein.

Auch hier können wir wieder alle Fälle durchgehen. Wir erhalten dann die folgende Tabelle (der Leser möge sich anhand des Beispiels selbst davon überzeugen):

Eingabe 1	**Eingabe 2**	**Ergebnis**
falsch	falsch	falsch

Eingabe 1	Eingabe 2	Ergebnis
falsch	wahr	wahr
wahr	falsch	wahr
wahr	wahr	wahr

Eine ODER-Verknüpfung in C wird durch den |-Operator repräsentiert. Beispiel:

```
int a;
a = 45 | 35
```

Eine weitere Verknüpfung ist XOR bzw. XODER (exklusives Oder), die auch als Antivalenz bezeichnet wird. Eine Antivalenzbedingung ist genau dann wahr, wenn die Bedingungen antivalent sind, das heißt, wenn A und B unterschiedliche Wahrheitswerte besitzen (siehe dazu untenstehende Wahrheitstabelle).

Man kann sich die XOR-Verknüpfung auch an folgendem Beispiel klar machen:

Entwederheute **oder**morgen gehe ich einkaufen

Hier lässt sich auf die gleiche Weise wie oben die Wahrheitstabelle herleiten:

Eingabe 1	Eingabe 2	Ergebnis
falsch	falsch	falsch
falsch	wahr	wahr
wahr	falsch	wahr
wahr	wahr	falsch

Ein XOR–Verknüpfung in C wird durch den ^-Operator repräsentiert. Beispiel:

```
int a;
a = a ^ 35 // in Kurzschreibweise: a ^= 35
```

Es gibt insgesamt 2^4=16 mögliche Verknüpfungen. Dies entspricht der Anzahl der möglichen Kombinationen der Spalte c in der Wahrheitstabelle. Ein Beispiel für eine solche Verknüpfung, die C nicht kennt, ist die Äquivalenzverknüpfung. Will man diese Verknüpfung erhalten, so muss man entweder eine Funktion schreiben, oder auf die boolsche Algebra zurückgreifen. Dies würde aber den Rahmen dieses Buches sprengen und soll deshalb hier nicht erläutert werden.

Eine weitere Möglichkeit, die einzelnen Bits zu beeinflussen, ist der Komplement-Operator. Mit ihm wird der Wahrheitswert aller Bits umgedreht:

Eingabe	Ergebnis
falsch	wahr
wahr	falsch

Das Komplement wird in C durch den ~-Operator repräsentiert. Beispiel:

```
int a;
a = ~45
```

Wie beim Rechnen mit den Grundrechenarten gibt es auch bei den boolschen Operatoren einen Vorrang. Den höchsten Vorrang hat der Komplement-Operator, gefolgt vom UND-Operator und dem XOR-Operator und schließlich dem ODER-Operator. So entspricht beispielsweise

```
a | b & ~c
```

der geklammerten Fassung

```
a | (b & (~c))
```

Es fragt sich nun, wofür solche Verknüpfungen gut sein sollen. Dies wollen wir an zwei Beispielen zeigen (wobei wir in diesem Beispiel von einem Integer mit 16 Bit ausgehen). Bei den Zahlen 0010 1001 0010 1001 und 0111 0101 1001 1100 wollen wir Bit zwei setzen (Hinweis: Normalerweise wird ganz rechts mit 0 beginnend gezählt). Alle anderen Bits sollen unberührt von der Veränderung bleiben. Wie erreichen wir das? Ganz einfach: Wir verknüpfen die Zahlen jeweils durch eine Oder-Verknüpfung mit 0000 0000 0000 0100. Wie Sie im folgenden sehen, erhalten wird dadurch tatsächlich das richtige Ergebnis:

```
0010 1001 0010 1001
0000 0000 0000 0100
0010 1001 0010 1101
```

Prüfen Sie das Ergebnis anhand der Oder-Wahrheitstabelle nach! Tatsächlich bleiben alle anderen Bits unverändert. Und was, wenn das zweite Bit bereits gesetzt ist? Sehen wir es uns an:

```
0111 0101 1001 1100
0000 0000 0000 0100
0111 0101 1001 1100
```

Auch hier klappt alles wie erwartet, so dass wir annehmen dürfen, dass dies auch bei jeder anderen Zahl funktioniert.

Wir stellen uns nun die Frage, ob Bit fünf gesetzt ist oder nicht. Für uns ist dies sehr einfach, da wir nur ablesen müssen. Die Rechnerhardware hat diese Fähigkeit aber leider nicht. Wir müssen deshalb auch in diesem Fall zu einer Verknüpfung greifen: Wenn wir eine beliebige Zahl durch eine Und–Verknüpfung mit 0000 0000 0010 0000 verknüpfen, so muss das Ergebnis, wenn Bit fünf gesetzt ist, einen Wert ungleich null ergeben, andernfalls muss das Ergebnis gleich null sein.

Wir nehmen nochmals die Zahlen 0010 1001 0010 1001 und 0111 0101 1001 1100 für unser Beispiel:

```
0010 1001 0010 1001
0000 0000 0010 0000
0000 0000 0010 0000
```

Da das Ergebnis ungleich null ist, können wir darauf schließen, dass das Bit gesetzt ist. Sehen wir uns nun das zweite Beispiel an, in dem das fünfte Bit nicht gesetzt ist:

```
0111 0101 1001 1100
0000 0000 0010 0000
0000 0000 0000 0000
```

Das Ergebnis ist nun gleich null, daher wissen wir, dass das fünfte Bit nicht gesetzt sein kann. Über eine Abfrage, wie wir sie im nächsten Kapitel kennenlernen werden, könnten wir das Ergebnis für unseren Programmablauf benutzen.

5.7 Bedingungsoperator

Der Bedingungsoperator liefert abhängig von einer Bedingung einen von zwei möglichen Ergebniswerten. Er hat drei Operanden: Die Bedingung, den Wert für den Fall, dass die Bedingung zutrifft und den Wert für den Fall dass sie nicht zutrifft. Die Syntax ist

```
bedingung ? wert_wenn_wahr : wert_wenn_falsch
```

Für eine einfache **if**-Anweisung wie die folgende:

```
/* Falls a größer als b ist, wird a zurückgegeben, ansonsten b. */
if (a > b)
   return a;
 else
   return b;
```

kann daher kürzer geschrieben werden

```
return (a > b) ? a : b;
/* Falls a größer als b ist, wird a zurückgegeben, ansonsten b. */
```

Der Bedingungsoperator ist nicht, wie oft angenommen, eine verkürzte Schreibweise für if-else. Die wichtigsten Unterschiede sind:

- Der Bedingungsoperator hat im Gegensatz zu if-else einen Ergebniswert und kann daher z.B. in Formeln und Funktionsaufrufen verwendet werden
- Bei if-Anweisungen kann der else-Teil entfallen, der Bedingungsoperator verlangt stets eine Angabe von beiden Ergebniswerten

Selbstverständlich können Ausdrücke mit diesem Operator beliebig geschachtelt werden. Das Maximum von drei Zahlen erhalten wir beispielsweise so:

```
return a > b ? (a > c ? a : c) : (b > c ? b : c);
```

An diesem Beispiel sehen wir auch sofort einen Nachteil des Bedingungsoperators: Es ist sehr unübersichtlich, verschachtelten Code mit ihm zu schreiben.

6 Kontrollstrukturen

Bisher haben unsere Programme einen streng linearen Ablauf gehabt. In diesem Kapitel werden Sie lernen, wie Sie den Programmfluss steuern können.

6.1 Bedingungen

Um auf Ereignisse zu reagieren, die erst bei der Programmausführung bekannt sind, werden Bedingungsanweisungen eingesetzt. Eine Bedingungsanweisung wird beispielsweise verwendet, um auf Eingaben des Benutzers reagieren zu können. Je nachdem, was der Benutzer eingibt, ändert sich der Programmablauf.

6.1.1 if

Beginnen wir mit der `if`-Anweisung. Sie hat die folgende Syntax:

```
if(expression) statement;
```

Optional kann eine alternative Anweisung angegeben werden, wenn die Bedingung `expression`nicht erfüllt wird:

```
if(expression)
  statement;
else
  statement;
```

Mehrere Fälle müssen verschachtelt abgefragt werden:

```
if(expression1)
  statement;
else
  if(expression2)
    statement;
  else
    statement;
```

Hinweis: `else if`- und `else`-Anweisungen sind optional.

Wenn der Ausdruck (engl. expression) nach seiner Auswertung wahr ist, d.h. von Null(0) verschieden, so wird die folgende Anweisung bzw. der folgende Anweisungsblock ausgeführt (statement). Ist der Ausdruck gleich Null und somit die Bedingungen nicht erfüllt, wird der `else`-Zweig ausgeführt, sofern vorhanden.

Klingt kompliziert, deshalb werden wir uns dies nochmals an zwei Beispielen ansehen:

```
#include <stdio.h>

int main(void)
{
  int zahl;
  printf("Bitte eine Zahl >5 eingeben: ");
  scanf("%i", &zahl);

  if(zahl > 5)
    printf("Die Zahl ist größer als 5\n");

  printf("Tschüß! Bis zum nächsten mal\n");

  return 0;
}
```

Wir nehmen zunächst einmal an, dass der Benutzer die Zahl 7 eingibt. In diesem Fall ist der Ausdruck `zahl > 5`true (wahr) und liefert eine 1 zurück. Da dies ein Wert ungleich 0 ist, wird die auf `if`folgende Zeile ausgeführt und "Die Zahl ist größer als 5" ausgegeben. Anschließend wird die Bearbeitung mit der Anweisung `printf("Tschüß! Bis zum nächsten mal\n")`fortgesetzt .

Wenn wir annehmen, dass der Benutzer eine 3 eingegeben hat, so ist der Ausdruck `zahl > 5`false (falsch) und liefert eine 0 zurück. Deshalb wird `printf("Die Zahl ist größer als 5")`nicht ausgeführt und nur "Tschüß! Bis zum nächsten mal" ausgegeben.

Wir können die `if`-Anweisung auch einfach lesen als: "Wenn `zahl`größer als 5 ist, dann gib "Die Zahl ist größer als 5" aus". In der Praxis wird man sich keine Gedanken machen, welches Resultat der Ausdruck `zahl > 5`hat.

Das zweite Beispiel, das wir uns ansehen, besitzt neben `if`auch ein `else if`und ein `else`:

```
#include <stdio.h>

int main(void)
{
  int zahl;
  printf("Bitte geben Sie eine Zahl ein: ");
  scanf("%d", &zahl);

  if(zahl > 0)
    printf("Positive Zahl\n");
  else if(zahl < 0)
    printf("Negative Zahl\n");
  else
    printf("Zahl gleich Null\n");

  return 0;
}
```

Nehmen wir an, dass der Benutzer die Zahl -5 eingibt. Der Ausdruck `zahl > 0`ist in diesem Fall falsch, weshalb der Ausdruck ein false liefert (was einer 0 entspricht). Deshalb wird die darauffolgende Anweisung nicht ausgeführt. Der Ausdruck `zahl < 0`ist dagegen erfüllt, was wiederum bedeutet, dass der Ausdruck wahr ist (und damit eine 1 liefert) und so die folgende Anweisung ausgeführt wird.

Nehmen wir nun einmal an, der Benutzer gibt eine 0 ein. Sowohl der Ausdruck `zahl > 0`als auch der Ausdruck `zahl < 0`sind dann nicht erfüllt. Der `if`- und der `if - else`-Block werden deshalb nicht ausgeführt. Der Compiler trifft anschließend allerdings auf die `el-`

se-Anweisung. Da keine vorherige Bedingung zutraf, wird die anschließende Anweisung ausgeführt.

Wir können die `if - else if - else`–Anweisung auch lesen als: "Wenn `zahl`größer ist als 0, gib "Positive Zahl" aus, ist `zahl`kleiner als 0, gib "Negative Zahl" aus, ansonsten gib "Zahl gleich Null" aus."

Fassen wir also nochmals zusammen: Ist der Ausdruck in der `if`oder `if - else`-Anweisung erfüllt (wahr), so wird die nächste Anweisung bzw. der nächste Anweisungsblock ausgeführt. Trifft keiner der Ausdrücke zu, so wird die Anweisung bzw. der Anweisungsblock, die `else`folgen, ausgeführt.

Es wird im Allgemeinen als ein guter Stil angesehen, jede Verzweigung einzeln zu klammern. So sollte man der Übersichtlichkeit halber das obere Beispiel so schreiben:

```
#include <stdio.h>

int main(void)
{
  int zahl;
  printf("Bitte geben Sie eine Zahl ein: ");
  scanf("%d", &zahl);

  if(zahl > 0) {
    printf("Positive Zahl\n");
  } else if(zahl < 0) {
    printf("Negative Zahl\n");
  } else {
    printf("Zahl gleich Null\n");
  }

  return 0;
}
```

Versehentliche Fehler wie

```
int a;

if(zahl > 0)
  a = berechne_a(); printf("Der Wert von a ist %d\n", a);
```

was so verstanden werden würde

```
int a;

if(zahl > 0) {
  a = berechne_a();
}

printf("Der Wert von a ist %d\n", a);
```

werden so vermieden.

6.1.2 Bedingter Ausdruck

Mit dem bedingten Ausdruck kann man eine `if- else`-Anweisung wesentlich kürzer formulieren. Sie hat die Syntax

```
exp1 ? exp2 : exp3
```

Zunächst wird das Ergebnis von `exp1`ermittelt. Liefert dies einen Wert ungleich 0 und ist somit true, dann ist der Ausdruck `exp2`das Resultat der bedingten Anweisung, andernfalls ist `exp3`das Resultat.

Beispiel:

```
int x = 20;
x = (x >= 10) ? 100 : 200;
```

Der Ausdruck `x >= 10`ist wahr und liefert deshalb eine 1. Da dies ein Wert ungleich 0 ist, ist das Resultat des bedingten Ausdrucks 100.

Der obige bedingte Ausdruck entspricht

```
if(x >= 10)
  x = 100;
else
  x = 200;
```

Die Klammern in unserem Beispiel sind nicht unbedingt notwendig, da Vergleichsoperatoren einen höheren Vorrang haben als der `?:`-Operator. Allerdings werden sie von vielen Programmierern verwendet, da sie die Lesbarkeit verbessern.

Der bedingte Ausdruck wird häufig, aufgrund seines Aufbaus, ternärer bzw. dreiwertiger Operator genannt.

6.1.3 switch

Eine weitere Auswahlanweisung ist die `switch`-Anweisung. Sie wird in der Regel verwendet, wenn eine unter vielen Bedingungen ausgewählt werden soll. Sie hat die folgende Syntax:

```
switch(expression)
{
  case const-expr: statements
  case const-expr: statements
  ...
  default: statements
}
```

In den runden Klammern der `switch`-Anweisung steht der Ausdruck, welcher mit den Konstanten (`const-expr`) verglichen wird, die den `case`-Anweisungen direkt folgen. War ein Vergleich positiv, wird zur entsprechenden `case`-Anweisung gesprungen und sämtlicher darauffolgender Code ausgeführt (eventuelle weitere `case`-Anweisungen darin sind wirkungslos). Eine `break`-Anweisung beendet die `switch`-Verzweigung und setzt bei der Anweisung nach der schließenden geschweiften Klammer fort. Optional kann eine `default`-Anweisung angegeben werden, zu der gesprungen wird, falls keiner der Vergleichswerte passt.

Vorsicht: Im Gegensatz zu anderen Programmiersprachen bricht die `switch`-Anweisung nicht ab, wenn eine `case`-Bedingung erfüllt ist. Eine `break`-Anweisung ist zwingend erforderlich, wenn die nachfolgenen `case`-Blöcke nicht bearbeitet werden sollen.

Sehen wir uns dies an einem textbasierenden Rechner an, bei dem der Benutzer durch die Eingabe eines Zeichens eine der Grundrechenarten auswählen kann:

```
#include <stdio.h>

int main(void)
{
  double zahl1, zahl2;
  char auswahl;
  printf("\nMini-Taschenrechner\n");
  printf("------------------\n\n");

  do
  {
     printf("\nBitte geben Sie die erste Zahl ein: ");
     scanf("%lf", &zahl1);
     printf("Bitte geben Sie die zweite Zahl ein: ");
     scanf("%lf", &zahl2);
     printf("\nZahl (a) addieren, (s) subtrahieren, (d) dividieren oder (m)
 multiplizieren?");
     printf("\nZum Beenden wählen Sie (b) ");
     scanf(" %c",&auswahl);

     switch(auswahl)
     {
       case 'a' :
       case 'A' :
         printf("Ergebnis: %lf", zahl1 + zahl2);
         break;
       case 's' :
       case 'S' :
         printf("Ergebnis: %lf", zahl1 - zahl2);
         break;
       case 'D' :
       case 'd' :
         if(zahl2 == 0)
           printf("Division durch 0 nicht möglich!");
         else
           printf("Ergebnis: %lf", zahl1 / zahl2);
         break;
       case 'M' :
       case 'm' :
         printf("Ergebnis: %lf", zahl1 * zahl2);
         break;
       case 'B' :
       case 'b' :
         break;
       default:
         printf("Fehler: Diese Eingabe ist nicht möglich!");
         break;
     }
   }

   while(auswahl != 'B' && auswahl != 'b');

   return 0;
}
```

Mit der `do-while`-Schleife wollen wir uns erst später[1] beschäftigen. Nur so viel: Sie dient dazu, dass der in den Blockklammern eingeschlossene Teil nur solange ausgeführt wird, bis der Benutzer `b`oder `B`zum Beenden eingegeben hat.

1 Kapitel 6.2.2 auf Seite 56

Die Variable `auswahl`erhält die Entscheidung des Benutzers für eine der vier Grundrechenarten oder den Abbruch des Programms. Gibt der Anwender beispielsweise ein kleines 's' ein, fährt das Programm bei der Anweisung `case('s')`fort und es werden solange alle folgenden Anweisungen bearbeitet, bis das Programm auf ein `break`stößt. Wenn keine der `case`Anweisungen zutrifft, wird die `default`-Anweisung ausgeführt und eine Fehlermeldung ausgegeben.

Etwas verwirrend mögen die Anweisungen `case('B')`und `case('b')`sein, denen unmittelbar `break`folgt. Sie sind notwendig, damit bei der Eingabe von B oder b nicht die `default`-Anweisung ausgeführt wird.

6.2 Schleifen

Schleifen werden verwendet, um einen Programmabschnitt mehrmals zu wiederholen. Sie kommen in praktisch jedem größeren Programm vor.

6.2.1 For-Schleife

Die `for`-Schleife wird in der Regel dann verwendet, wenn von vornherein bekannt ist, wie oft die Schleife durchlaufen werden soll. Die `for`-Schleife hat die folgende Syntax:

```
for (expression_opt; expression_opt; expression_opt)
statement
```

In der Regel besitzen `for`-Schleifen einen Schleifenzähler. Dies ist eine Variable, zu der bei jedem Durchgang ein Wert addiert oder subtrahiert wird (oder die durch andere Rechenoperationen verändert wird). Der Schleifenzähler wird über den ersten Ausdruck initialisiert. Mit dem zweiten Ausdruck wird überprüft, ob die Schleife fortgesetzt oder abgebrochen werden soll. Letzterer Fall tritt ein, wenn dieser den Wert 0 annimmt – also der Ausdruck false (falsch) ist. Der letzte Ausdruck dient schließlich dazu, den Schleifenzähler zu verändern.

Mit einem Beispiel sollte dies verständlicher werden. Das folgende Programm zählt von 1 bis 5:

```
#include <stdio.h>

int main()
{
   int i;

   for(i = 1; i <= 5; ++i)
     printf("%d  ", i);

   return 0;
}
```

Die Schleife beginnt mit dem Wert 1 (i = 1) und erhöht den Schleifenzähler i bei jedem Durchgang um 1 (++i). Solange der Wert i kleiner oder gleich 5 ist (i <= 5), wird die Schleife durchlaufen. Ist i gleich 6 und daher die Aussage i <= 5 falsch, wird der Wert 0

zurückgegeben und die Schleife abgebrochen. Insgesamt wird also die Schleife 5mal durchlaufen.

Wenn das Programm kompiliert und ausgeführt wird, erscheint die folgende Ausgabe auf dem Monitor:

```
1  2  3  4  5
```

Anstelle des Präfixoperators hätte man auch den Postfixoperator `i++`benutzen und `for(i = 1; i <= 5; i++)`schreiben können. Diese Variante unterscheidet sich nicht von der oben verwendeten. Eine weitere Möglichkeit wäre, `for(i = 1; i <= 5; i = i + 1)`oder `for(i = 1; i <= 5; i += 1)`zu schreiben. Die meisten Programmierer benutzen eine der ersten beiden Varianten, da sie der Meinung sind, dass schneller ersichtlich wird, dass `i`um eins erhöht wird und dass durch den Inkrementoperator Tipparbeit gespart werden kann.

Damit die `for`-Schleife noch etwas klarer wird, wollen wir uns noch ein paar Beispiele ansehen:

```
for(i = 0;  i < 7; i += 1.5)
```

Der einzige Unterschied zum letzten Beispiel besteht darin, dass die Schleife nun in 1,5er Schritten durchlaufen wird. Der nachfolgende Befehl oder Anweisungsblock wird insgesamt 5mal durchlaufen. Dabei nimmt der Schleifenzähler i die Werte 0, 1.5, 3, 4.5 und 6 an (Die Variable i muss hier natürlich einen Gleitkommadatentyp haben).

```
for(i = 20; i > 5; i -= 5)
```

Diesmal zählt die Schleife rückwärts. Sie wird dreimal durchlaufen. Der Schleifenzähler nimmt dabei die Werte 20, 15 und 10 an. Und noch ein letztes Beispiel:

```
for(i = 1; i < 20; i *= 2)
```

Prinzipiell lassen sich für die Schleife alle Rechenoperationen benutzen. In diesem Fall wird in der Schleife die Multiplikation benutzt. Sie wird 5mal durchlaufen. Dabei nimmt der Schleifenzähler die Werte 1, 2, 4, 8 und 16 an.

Wie Sie aus der Syntax unschwer erkennen können, sind die Ausdrücke in den runden Klammern optional. So ist beispielsweise

```
for(;;)
```

korrekt. Da nun der zweite Ausdruck immer wahr ist, und damit der Schleifenkopf niemals den Wert 0 annehmen kann, wird die Schleife unendlich oft durchlaufen. Eine solche Schleife wird auch als Endlosschleife bezeichnet, da sie niemals endet (in den meisten Betriebssystemen gibt es eine Möglichkeit das dadurch "stillstehende" Programm mit einer Tastenkombination abzubrechen). Endlosschleifen können beabsichtigt (siehe dazu auch weiter unten die break-Anweisung[2]) oder unbeabsichtigte Programmierfehler sein.

2 Kapitel 6.2.4 auf Seite 59

Mehrere Befehle hinter einer `for`-Anweisung müssen immer in Blockklammern eingeschlossen werden:

```
for(i = 1; i < 5; i++)
{
  printf("\nEine Schleife: ");
  printf("%d ", i);
}
```

Schleifen lassen sich auch schachteln, das heißt, innerhalb einer Schleife dürfen sich eine oder mehrere weitere Schleifen befinden. Beispiel:

```
#include <stdio.h>

int main()
{
   int i, j, Zahl = 1;

   for (i = 1; i <= 11; i++)
   {
      for (j = 1; j <= 10; j++)
      {
         printf ("%4i", Zahl++);
      }
      printf ("\n");
   }

   return 0;
}
```

Nach der Kompilierung und Übersetzung des Programms erscheint die folgende Ausgabe:

```
   1   2   3   4   5   6   7   8   9  10
  11  12  13  14  15  16  17  18  19  20
  21  22  23  24  25  26  27  28  29  30
  31  32  33  34  35  36  37  38  39  40
  41  42  43  44  45  46  47  48  49  50
  51  52  53  54  55  56  57  58  59  60
  61  62  63  64  65  66  67  68  69  70
  71  72  73  74  75  76  77  78  79  80
  81  82  83  84  85  86  87  88  89  90
  91  92  93  94  95  96  97  98  99 100
 101 102 103 104 105 106 107 108 109 110
```

Damit bei der Ausgabe alle 10 Einträge eine neue Zeile beginnt, wird die innere Schleife nach 10 Durchläufen beendet. Anschließend wird ein Zeilenumbruch ausgegeben und die innere Schleife von der äußeren Schleife wiederum insgesamt 11-mal aufgerufen.

6.2.2 While-Schleife

Häufig kommt es vor, dass eine Schleife, beispielsweise bei einem bestimmten Ereignis, abgebrochen werden soll. Ein solches Ereignis kann z.B. die Eingabe eines bestimmen Wertes sein. Hierfür verwendet man meist die `while`-Schleife, welche die folgende Syntax hat:

```
while (expression)
  statement
```

Im folgenden Beispiel wird ein Text solange von der Tastatur eingelesen, bis der Benutzer die Eingabe abschließt (In der Microsoft-Welt geschieht dies durch <Strg>-<Z>, in der UNIX-Welt über die Tastenkombination <Strg>-<D>). Als Ergebnis liefert das Programm die Anzahl der Leerzeichen:

```
#include <stdio.h>

int main()
{
  int c;
  int zaehler = 0;

  printf("Leerzeichenzähler - zum Beenden STRG + D / STRG + Z\n");

  while((c = getchar()) != EOF)
  {
    if(c == ' ')
      zaehler++;
  }

  printf("Anzahl der Leerzeichen: %d\n", zaehler);

  return 0;
}
```

Die Schleife wird abgebrochen, wenn der Benutzer die Eingabe (mit <Strg>-<Z> oder <Strg>-<D>) abschließt und somit das nächste zu liefernde Zeichen das EOF-Zeichen ist. In diesem Fall ist der Ausdruck `((c = getchar()) != EOF)`nicht mehr wahr, liefert 0 zurück, und die Schleife wird beendet.

Bitte beachten Sie, dass die Klammer um `c = getchar()`nötig ist, da der Ungleichheitsoperator eine höhere Priorität[3] hat als der Zuweisungsoperator `=`. Neben den Zuweisungsoperatoren besitzen auch die logischen Operatoren Und (`&`), Oder (`|`) sowie XOR (`^`) eine niedrigere Priorität.

Noch eine Anmerkung zu diesem Programm: Wie Sie vielleicht bereits festgestellt haben, wird das Zeichen, das `getchar()`zurückliefert, in einer Variable des Typs Integer gespeichert. Für die Speicherung eines Zeichenwertes genügt, wie wir bereits gesehen haben, eine Variable vom Typ Character. Der Grund dafür, dass wir dies hier nicht können, liegt im ominösen EOF-Zeichen. Es dient normalerweise dazu, das Ende einer Datei zu markieren - auf Englisch das End of File - oder kurz EOF. Allerdings ist EOF ein negativer Wert vom Typ `int`, so dass kein "Platz" mehr in einer Variable vom Typ `char`ist. Viele Implementierungen benutzen -1 um das EOF-Zeichen darzustellen, was der ANSI-C-Standard allerdings nicht vorschreibt (der tatsächliche Wert ist in der Headerdatei `<stdio.h>`[4]abgelegt).

Ersetzen einer for-Schleife

Eine `for`-Schleife kann immer durch eine `while`-Schleife ersetzt werden. So ist beispielsweise unser `for`-Schleifenbeispiel aus dem ersten Abschnitt[5] mit der folgenden `while`-Schleife äquivalent:

3 Kapitel 22.6.10 auf Seite 172
4 Kapitel 22.10.2 auf Seite 177
5 Kapitel 6.2.1 auf Seite 54

```
#include <stdio.h>

int main()
{
  int x = 1;

  while(x <= 5)
  {
    printf("%d  ", x);
    ++x;
  }

  return 0;
}
```

Ob man `while`oder `for`benutzt, hängt letztlich von der Vorliebe des Programmierers ab. In diesem Fall würde man aber vermutlich eher eine `for`-Schleife verwenden, da diese Schleife eine Zählervariable enthält, die bei jedem Schleifendurchgang um eins erhöht wird.

6.2.3 Do-While-Schleife

Im Gegensatz zur `while`-Schleife findet bei der Do-while-Schleife die Überprüfung der Wiederholungsbedingung am Schleifenende statt. So kann garantiert werden, dass die Schleife mindestens einmal durchlaufen wird. Sie hat die folgende Syntax:

```
do
  statement
while (expression);
```

Das folgende Programm addiert solange Zahlen auf, bis der Anwender eine 0 eingibt:

```
#include <stdio.h>

int main(void)
{
 float zahl;
 float ergebnis = 0;

 do
 {
   printf ("Bitte Zahl zum Addieren eingeben (0 zum Beenden):");
   scanf("%f",&zahl);
   ergebnis += zahl;
 }
 while (zahl != 0);

 printf("Das Ergebnis ist %f \n", ergebnis);

 return 0;
}
```

Die Überprüfung, ob die Schleife fortgesetzt werden soll, findet in Zeile 14 statt. Mit `do`in Zeile 8 wird die Schleife begonnen, eine Prüfung findet dort nicht statt, weshalb der Block von Zeile 9 bis 13 in jedem Fall mindestens einmal ausgeführt wird.

Wichtig: Beachten Sie, dass das `while`mit einem Semikolon abgeschlossen werden muss, sonst wird das Programm nicht korrekt ausgeführt!

6.2.4 Schleifen abbrechen

continue

Eine `continue`-Anweisung beendet den aktuellen Schleifendurchlauf und setzt, sofern die Schleifen-Bedingung noch erfüllt ist, beim nächsten Durchlauf fort.

```
#include <stdio.h>

int main(void)
{
  double i;

  for(i = -10; i <= 10; i++)
  {
    if(i == 0)
     continue;

    printf("%lf \n", 1/i);
  }

  return 0;
}
```

Das Programm berechnet in ganzzahligen Schritten die Werte für `1/i`im Intervall [-10, 10]. Da die Division durch Null nicht erlaubt ist, springen wir mit Hilfe der `if`-Bedingung wieder zum Schleifenkopf.

break

Die `break`-Anweisung beendet eine Schleife und setzt bei der ersten Anweisung **nach**der Schleife fort. Nur innerhalb einer Wiederholungsanweisung, wie in for-, while-, do-while-Schleifen oder innerhalb einer switch-Anweisung ist eine break-Anweisung funktionsfähig. Sehen wir uns dies an folgendem Beispiel an:

```
#include <stdio.h>

int eingabe;
int passwort = 2323;

int main(void) {
    while (1) {
        printf("Geben Sie bitte das Zahlen-Passwort ein: ");
        scanf("%d", &eingabe);

        if (passwort == eingabe) {
            printf("Passwort korrekt\n");
            break;
        } else {
            printf("Das Passwort ist nicht korrekt.\n");
        }

        printf("Bitte versuchen Sie es nochmal!\n");
    }

    printf("Programm beendet\n");

    return 0;
}
```

Wie Sie sehen ist die while-Schleife als Endlosschleife konzipiert. Hat man das richtige Passwort eingegeben, so wird die printf-Anweisung ausgegeben, und anschließend wird diese Endlosschleife durch die break-Anweisung verlassen. Die nächste Anweisung, die dann ausgeführt wird, ist die printf-Anweisung unmittelbar nach der Schleife. Ist das Passwort aber inkorrekt, so wird der else-Block mit den weiteren printf-Anweisungen in der while-Schleife ausgeführt. Anschließend wird die while-Schleife wieder ausgeführt.

Tastaturpuffer leeren

Es ist wichtig, den Tastaturpuffer zu leeren, damit Tastendrücke nicht eine unbeabsichtigte Aktion auslösen (Es besteht außerdem noch die Gefahr eines Puffer-Überlaufs). In ANSI-C-Compilern bzw. deren Laufzeitbibliothek ist die Vollpufferung die Standardeinstellung; diese ist auch sinnvoller als keine Pufferung, da dadurch weniger Schreib- und Leseoperationen stattfinden. Die Puffergröße ist abhängig vom Compiler; in der Regel liegt sie meistens bei 256 KiB, 512 KiB, 1024 KiB oder 4096 KiB. Die genaue Größe ist in der Headerdatei von <stdio.h> mit der Konstanten `Bufsiz`deklariert. Weiteres zu Pufferung und `setbuf()`/`setvbuf()`wird in den weiterführenden Kapiteln behandelt.

Sehen wir uns dies an einem kleinen Spiel an: Der Computer ermittelt eine Zufallszahl zwischen 1 und 100, die der Nutzer dann erraten soll. Dabei gibt es immer einen Hinweis, ob die Zahl kleiner oder größer als die eingegebene Zahl ist.

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

int main(void)
{
  int zufallszahl,  eingabe;
  int durchgaenge;
  char auswahl;
  srand(time(0));

  printf("\nLustiges Zahlenraten");
  printf("\n--------------------");
  printf("\nErraten Sie die Zufallszahl in moeglichst wenigen Schritten!");
  printf("\nDie Zahl kann zwischen 1 und 100 liegen");

  do
  {
    zufallszahl = (rand() % 100 + 1);
    durchgaenge = 1;

    while(1)
    {
      printf("\nBitte geben Sie eine Zahl ein: ");
      scanf("%d", &eingabe);

      if(eingabe > zufallszahl)
      {
        printf("Leider falsch! Die zu erratende Zahl ist kleiner");
        durchgaenge++;
      }
      else if(eingabe < zufallszahl)
      {
        printf("Leider falsch! Die zu erratende Zahl ist größer");
        durchgaenge++;
      }
```

```
      else
      {
        printf("Glückwunsch! Sie haben die Zahl in %d", durchgaenge);
        printf(" Schritten erraten.");
        break;
      }
    }

    printf("\nNoch ein Spiel? (J/j für weiteres Spiel)");

    /* Rest vom letzten scanf aus dem Tastaturpuffer löschen */
    while((auswahl = getchar()) != '\n' && auswahl != EOF);

    auswahl = getchar();

  } while(auswahl == 'j' || auswahl == 'J');

  return 0;
}
```

Wie Sie sehen, ist die innere `while`-Schleife als Endlosschleife konzipiert. Hat der Spieler die richtige Zahl erraten, so wird der `else`-Block ausgeführt. In diesem wird die Endlosschleife schließlich mit `break`abgebrochen. Die nächste Anweisung, die dann ausgeführt wird, ist die `printf`-Anweisung unmittelbar nach der Schleife.

Die äußere `while`-Schleife in Zeile 52 wird solange wiederholt, bis der Benutzer nicht mehr mit einem kleinen oder großen `j`antwortet. Beachten Sie, dass im Gegensatz zu den Operatoren `&`und `|`die Operatoren `&&`und `||`streng von links nach rechts bewertet werden.

In diesem Beispiel hat dies keine Auswirkungen. Allerdings schreibt der Standard für den `||`-Operator auch vor, dass, wenn der erste Operand des Ausdrucks verschieden von 0 (wahr) ist, der Rest nicht mehr ausgewertet wird. Die Folgen soll dieses Beispiel verdeutlichen:

```
int c, a = 5;

while (a == 5 || (c = getchar()) != EOF)
```

Da der Ausdruck `a == 5`true ist, liefert er also einen von 0 verschiedenen Wert zurück. Der Ausdruck `c = getchar()`wird deshalb erst gar nicht mehr ausgewertet, da bereits nach der Auswertung des ersten Operanden feststeht, dass die ODER-Verknüpfung den Wahrheitswert true besitzen muss (Wenn Ihnen dies nicht klar geworden ist, sehen Sie sich nochmals die Wahrheitstabelle der ODER-Verknüpfung[6] an). Dies hat zur Folge, dass `getchar()`nicht mehr ausgeführt und deshalb kein Zeichen eingelesen wird. Wenn wir wollen, dass `getchar()`aufgerufen wird, so müssen wir die Reihenfolge der Operanden umdrehen.

Dasselbe gilt natürlich auch für den `&&`-Operator, nur dass in diesem Fall der zweite Operand nicht mehr ausgewertet wird, wenn der erste Operand bereits 0 ist.

Beim `||`und `&&`-Operator handelt es sich um einen Sequenzpunkt: Wie wir gesehen haben, ist dies ein Punkt, bis zu dem alle Nebenwirkungen vom Compiler ausgewertet sein müssen. Auch hierzu ein Beispiel:

```
i = 7;
```

6 Kapitel 5.6 auf Seite 44

```
if(i++ == 5 || (i += 3) == 4)
```

Zunächst wird der erste Operand ausgewertet (`i++ == 5`) - es wird `i`um eins erhöht und mit dem Wert 5 verglichen. Wie wir gerade gesehen haben, wird der zweite Operand (`(i += 3) == 4`) nur dann ausgewertet, wenn feststeht, dass der erste Operand 0 liefert (bzw. keinen nicht von 0 verschiedenen Wert). Da der erste Operand keine wahre Aussage darstellt (i++ ergibt 8 zurück, wird dann auf Gleichheit mit 5 überprüft, gibt "falsch" zurück, da 8 nicht gleich 5 ist) wird der zweite ausgewertet. Hierbei wird zunächst i um 3 erhöht, das Ergebnis der Zuweisung (11) dann mit 4 verglichen. Es wird also der gesamte Ausdruck ausgewertet (er ergibt insgesamt übrigens "falsch", da weder der erste noch der zweite Operand "wahr" ergeben; 8 ist ungleich 5, und 11 ist ungleich 4). Die Auswertung findet auf jeden Fall in dieser Reihenfolge statt, nicht umgekehrt. Es ist also nicht möglich, dass zu i zuerst die 3 addiert wird und so den Wert 10 annimmt, um anschließend um 1 erhöht zu werden. Diese Tatsache ändert in diesem Beispiel nichts an der Falschheit des gesamten Ausdruckes, kann aber zu unbedachten Resultaten führen, wenn im zweiten Operator eine Funktion aufgerufen wird, die Nebenwirkungen hat (beispielsweise das Anlegen einer Datei). Ergibt der erste Operand einen Wert ungleich 0 (also wahr), so wird der zweite (rechts vom `||`-Operator) nicht mehr aufgerufen und die Datei nicht mehr angelegt.

Bevor wir uns weiter mit Kontrollstrukturen beschäftigen, lassen Sie uns aber noch einen Blick auf den Zufallsgenerator werfen, da er eine interessante Anwendung für den Modulo–Operator[7] darstellt. Damit der Zufallsgenerator nicht immer die gleichen Zahlen ermittelt, muss zunächst der Zufallsgenerator über `srand(time(0))`mit der Systemzeit initialisiert werden (wenn Sie diese Bibliotheksfunktionen in Ihrem Programm benutzen wollen, beachten Sie, dass Sie für die Funktion `time(0)`die Headerdatei `<time.h>`und für die Benutzung des Zufallsgenerators die Headerdatei `<stdlib.h>`einbinden müssen). Aber wozu braucht man nun den Modulo-Operator? Die Funktion `rand()`liefert einen Wert zwischen 0 und mindestens 32767. Um nun einen Zufallswert zwischen 1 und 100 zu erhalten, teilen wir den Wert durch hundert und addieren 1. Den Rest, der ja nun zwischen eins und hundert liegen muss, verwenden wir als Zufallszahl.

Bitte beachten Sie, dass `rand()`in der Regel keine sehr gute Streuung liefert. Für statistische Zwecke sollten Sie deshalb nicht auf die Standardbibliothek zurückgreifen.

6.3 Sonstiges

6.3.1 goto

Mit einer `goto`-Anweisung setzt man die Ausführung des Programms an einer anderen Stelle des Programms fort. Diese Stelle im Programmcode wird mit einem sogenannten Label definiert:

```
LabelName:
```

Zu einem Label springt man mit

7 Kapitel 1.5 auf Seite 7

```
goto LabelName;
```

In der Anfangszeit der Programmierung wurde `goto`[8] anstelle der eben vorgestellten Kontrollstrukturen verwendet. Das Ergebnis war eine sehr unübersichtliche Programmstruktur, die auch häufig als Spaghetticode[9] bezeichnet wurde. Bis auf wenige Ausnahmen ist es möglich, auf die `goto`-Anweisung zu verzichten (neuere Sprachen wie Java[10] kennen sogar überhaupt kein `goto`mehr). Einige der wenigen Anwendungsgebiete von `goto`werden Sie im Kapitel Programmierstil[11] finden, darüber hinaus werden Sie aber keine weiteren Beispiele in diesem Buch finden.

en:C Programming/Control[12] et:Programmeerimiskeel C/Keelestruktuurid[13] fi:C/Ohjausrakenteet[14] pl:C/Instrukcje sterujące[15]

8 http://de.wikipedia.org/wiki/goto%20
9 http://de.wikipedia.org/wiki/Spaghetticode%20
10 http://de.wikipedia.org/wiki/Java%20%28Programmiersprache%29
11 Kapitel 20.3 auf Seite 147
12 http://en.wikibooks.org/wiki/C%20Programming%2FControl
13 http://et.wikibooks.org/wiki/Programmeerimiskeel%20C%2FKeelestruktuurid
14 http://fi.wikibooks.org/wiki/C%2FOhjausrakenteet
15 http://pl.wikibooks.org/wiki/C%2FInstrukcje%20steruj%C4%85ce

7 Funktionen

Eine wichtige Forderung der strukturierten Programmierung ist die Vermeidung von Sprüngen innerhalb des Programms. Wie wir gesehen haben, ist dies in allen Fällen mit Kontrollstrukturen möglich.

Die zweite Forderung der strukturierten Programmierung ist die Modularisierung. Dabei wird ein Programm in mehrere Programmabschnitte, die Module zerlegt. In C werden solche Module auch als Funktionen bezeichnet. Andere Programmiersprachen bezeichnen Module als Unterprogramme oder unterscheiden zwischen Funktionen (Module mit Rückgabewert) und Prozeduren (Module ohne Rückgabewert). Trotz dieser unterschiedlichen Bezeichnungen ist aber dasselbe gemeint.

Objektorientierte Programmiersprachen gehen noch einen Schritt weiter und verwenden Klassen zur Modularisierung. Vereinfacht gesagt bestehen Klassen aus Methoden (vergleichbar mit Funktionen) und Attributen (Variablen). C selbst unterstützt keine Objektorientierte Programmierung, im Gegensatz zu C++, das auf C aufbaut.

Die Modularisierung hat eine Reihe von Vorteilen:

Bessere Lesbarkeit

Der Quellcode eines Programms kann schnell mehrere tausend Zeilen umfassen. Beim Linux Kernel sind es sogar über 15 Millionen Zeilen und Windows, das ebenfalls zum Großteil in C geschrieben wurde, umfasst schätzungsweise auch mehrere Millionen Zeilen. Um dennoch die Lesbarkeit des Programms zu gewährleisten, ist die Modularisierung unerlässlich.

Wiederverwendbarkeit

In fast jedem Programm tauchen die gleichen Problemstellungen mehrmals auf. Oft gilt dies auch für unterschiedliche Applikationen. Da nur Parameter und Rückgabetyp für die Benutzung einer Funktion bekannt sein müssen, erleichtert dies die Wiederverwendbarkeit. Um die Implementierungsdetails muss sich der Entwickler dann nicht mehr kümmern.

Wartbarkeit

Fehler lassen sich durch die Modularisierung leichter finden und beheben. Darüber hinaus ist es leichter, weitere Funktionalitäten hinzuzufügen oder zu ändern.

7.1 Funktionsdefinition

Im Kapitel Was sind Variablen[1] haben wir die Quaderoberfläche berechnet. Nun wollen wir eine Funktion schreiben, die eine ähnliche Aufgabe für uns übernimmt: die Berechnung

1 Kapitel 2.1 auf Seite 11

der Oberfläche eines Zylinders. Dazu schauen wir uns zunächst die Syntax einer Funktion an:

```
Rückgabetyp  Funktionsname(Parameterliste)
{
   Anweisungen
}
```

Die Anweisungen werden auch als Funktionsrumpf bezeichnet, die erste Zeile als Funktionskopf.

Wenn wir eine Funktion zur Zylinderoberflächenberechnung schreiben und diese benutzen sieht unser Programm wie folgt aus:

```
#include <stdio.h>

float zylinder_oberflaeche(float h, float r)
{
  float o;
  o=2*3.141*r*(r+h);
  return(o);
}

int main(void)
{
  float r,h;
  printf("Programm zur Berechnung einer Zylinderoberfläche");
  printf("\n\nHöhe des Zylinders: ");
  scanf("%f",&h);
  printf("\nRadius des Zylinders: ");
  scanf("%f",&r);
  printf("Oberfläche: %f \n",zylinder_oberflaeche(h,r));
  return 0;
}
```

- In Zeile 3 beginnt die Funktionsdefinition. Das `float`ganz am Anfang der Funktion, der sogenannte Funktionstyp, sagt dem Compiler, dass ein Wert mit dem Typ `float`zurückgegeben wird. In Klammern werden die Übergabeparameter `h`und `r`deklariert, die der Funktion übergeben werden.
- Mit `return`wird die Funktion beendet und ein Wert an die aufrufende Funktion zurückgegeben (hier: `main`). In unserem Beispiel geben wir den Wert von `o`zurück, also das Ergebnis unserer Berechnung. Der Datentyp des Ausdrucks muss mit dem Typ des Rückgabewertes des Funktionskopfs übereinstimmen. Würden wir hier beispielsweise versuchen, den Wert einer int-Variable zurückzugeben, würden wir vom Compiler eine Fehlermeldung erhalten.
  Soll der aufrufenden Funktion kein Wert zurückgegeben werden, muss als Typ der Rückgabewert `void`angegeben werden. Eine Funktion, die lediglich einen Text ausgibt hat beispielsweise den Rückgabetyp `void`, da sie keinen Wert zurückgibt.
- In Zeile 18 wird die Funktion `zylinder_oberflaeche`aufgerufen. Hier werden die beiden Parameter `h`und `r`übergeben. Der zurückgegebene Wert wird ausgegeben. Es wäre aber genauso denkbar, dass der Wert einer Variable zugewiesen, mit einem anderen Wert verglichen oder mit dem Rückgabewert weitergerechnet wird.
  Der Rückgabewert muss aber nicht ausgewertet werden. Es ist kein Fehler, wenn der Rückgabewert unberücksichtigt bleibt.

In unserem Beispiel haben wir den Rückgabetyp in `return`geklammert (Zeile 7). Die Klammerung ist aber optional und kann weggelassen werden (Zeile 19).

Auch die Funktion `main`hat einen Rückgabewert. Ist der Wert 0, so bedeutet dies, dass das Programm ordnungsgemäß beendet wurde, ist der Wert -1, so bedeutet dies, dass ein Fehler aufgetreten ist.

Jede Funktion muss einen Rückgabetyp besitzen. In der ursprünglichen Sprachdefinition von K&R wurde dies noch nicht gefordert. Wenn der Rückgabetyp fehlte, wurde defaultmäßig `int`angenommen. Dies ist aber inzwischen nicht mehr erlaubt. Jede Funktion muss einen Rückgabetyp explizit angeben.

Die folgenden Beispiele enthalten Funktionsdefinitionen, die einige typische Anfänger(denk)fehler zeigen:

```
void foo()
{
  /* Code */
  return 5; /* Fehler */
}
```

Eine Funktion, die als `void`deklariert wurde, darf keinen Rückgabetyp erhalten. Der Compiler sollte hier eine Fehlermeldung oder zumindest eine Warnung ausgeben.

```
int foo()
{
  /* Code */
  return 5;
  printf("Diese Zeile wird nie ausgeführt");
}
```

Bei diesem Beispiel wird der Compiler weder eine Warnung noch eine Fehlermeldung ausgeben. Allerdings wird die `printf`Funktion niemals ausgeführt, da `return`nicht nur einen Wert zurückgibt sondern die Funktion `foo()`auch beendet.

Das folgende Programm arbeitet hingegen völlig korrekt:

```
int max(int a, int b)
{
  if(a >= b){
    return a;
  }
  if(a < b){
    return b;
  }
}
```

Bei diesem Beispiel gibt der Compiler keine Fehlermeldung oder Warnung aus, da eine Funktion zwar nur einen Rückgabewert erhalten darf, aber mehrere `return`Anweisungen besitzen kann. In diesem Beispiel wird in Abhängigkeit der übergebenen Parameter entweder `a`oder `b`zurückgegeben.

7.2 Prototypen

Auch bei Funktionen unterscheidet man wie bei Variablen zwischen Definition und Deklaration. Mit

```
float zylinder_oberflaeche(float h, float r)
{
  float o;
  o=2*3.141*r*(r+h);
  return(o);
}
```

wird die Funktion `zylinder_oberflaeche`definiert.

Bei einer Funktionsdeklaration wird nur der Funktionskopf gefolgt von einem Semikolon angeben. Die Funktion `zylinder_oberflaeche`beispielsweise wird wie folgt deklariert:

```
float zylinder_oberflaeche(float h, float r);
```

Dies ist identisch mit

```
extern float zylinder_oberflaeche(float h, float r);
```

Die Meinungen, welche Variante benutzt werden soll, gehen hier auseinander: Einige Entwickler sind der Meinung, dass das Schlüsselwort `extern`die Lesbarkeit verbessert, andere wiederum nicht. Wir werden im Folgenden das Schlüsselwort `extern`in diesem Zusammenhang nicht verwenden.

Eine Trennung von Definition und Deklaration ist notwendig, wenn die Definition der Funktion erst nach der Benutzung erfolgen soll. Eine Deklaration einer Funktion wird auch als *Prototyp*oder *Funktionskopf*bezeichnet. Damit kann der Compiler überprüfen, ob die Funktion überhaupt existiert und Rückgabetyp und Typ der Argumente korrekt sind. Stimmen Prototyp und Funktionsdefinition nicht überein oder wird eine Funktion aufgerufen, die noch nicht definiert wurde oder keinen Prototyp besitzt, so ist dies ein Fehler.

Das folgende Programm ist eine weitere Abwandlung des Programms zur Berechnung der Zylinderoberfläche. Die Funktion `zylinder_oberflaeche`wurde dabei verwendet, bevor sie definiert wurde:

```
#include <stdio.h>

float zylinder_oberflaeche(float h, float r);

int main(void)
{
  float r,h,o;
  printf("Programm zur Berechnung einer Zylinderoberfläche");
  printf("\n\nHöhe des Zylinders:");
  scanf("%f",&h);
  printf("\nRadius des Zylinders:");
  scanf("%f",&r);
  printf("Oberfläche: %f \n",zylinder_oberflaeche(h,r));
  return 0;
}

float zylinder_oberflaeche(float h, float r)
{
  float o;
```

```
  o=2*3.141*r*(r+h);
  return(o);
}
```

Der Prototyp wird in Zeile 3 deklariert, damit die Funktion in Zeile 13 verwendet werden kann. An dieser Stelle kann der Compiler auch prüfen, ob der Typ und die Anzahl der übergebenen Parameter richtig ist (dies könnte er nicht, hätten wir keinen Funktionsprototyp deklariert). Ab Zeile 17 wird die Funktion `zylinder_oberflaeche`definiert.

Die Bezeichner der Parameter müssen im Prototyp und der Funktionsdefinition nicht übereinstimmen. Sie können sogar ganz weggelassen werden. So kann Zeile 3 auch ersetzt werden durch:

```
float zylinder_oberflaeche(float, float);
```

Wichtig: Bei Prototypen unterscheidet C zwischen einer leeren Parameterliste und einer Parameterliste mit `void`. Ist die Parameterliste leer, so bedeutet dies, dass die Funktion eine nicht definierte Anzahl an Parametern besitzt. Das Schlüsselwort `void`gibt an, dass der Funktion keine Werte übergeben werden dürfen. Beispiel:

```
int foo1();
int foo2(void);

int main(void)
{
  foo1(1, 2, 3); // kein Fehler
  foo2(1, 2, 3); // Fehler
  return 0;
}
```

Während der Compiler beim Aufruf der Funktion `foo1`in Zeile 6 keine Fehlermeldung ausgegeben wird, gibt der Compiler beim Aufruf der Funktion `foo2`in Zeile 7 eine Fehlermeldung aus. (Der Compiler wird höchstwahrscheinlich noch zwei weitere Warnungen oder Fehler ausgeben, da wir zwar Prototypen für die Funktionen `foo1`und `foo2`haben, die Funktion aber nicht definiert haben.)

Diese Aussage gilt übrigens nur für Prototypen: Laut C Standard bedeutet eine leere Liste bei Funktionsdeklarationen die Teil einer Definition sind, dass die Funktion keine Parameter hat. Im Gegensatz dazu bedeutet eine leere Liste in einer Funktionsdeklaration, die nicht Teil einer Definition sind (also Prototypen), dass keine Informationen über die Anzahl oder Typen der Parameter vorliegt - so wie wir das eben am Beispiel der Funktion `foo1`gesehen haben.

Noch ein Hinweis für Leser, die ihre C Programme mit einem C++ Compiler compilieren: Bei C++ würde auch im Fall von `foo1`eine Fehlermeldung ausgegeben, da dort auch eine leere Parameterliste bedeutet, dass der Funktion keine Parameter übergeben werden können.

Übrigens haben auch Bibliotheksfunktionen wie `printf`oder `scanf`einen Prototyp. Dieser befindet sich üblicherweise in der Headerdatei `stdio.h`oder anderen Headerdateien. Damit kann der Compiler überprüfen, ob die Anweisungen die richtige Syntax haben. Der Prototyp der `printf`Anweisung hat beispielsweise die folgende Form (oder ähnlich) in der `stdio.h`:

```
int printf(const char *, ...);
```

Findet der Compiler nun beispielsweise die folgende Zeile im Programm, gibt er einen Fehler aus:

```
printf(45);
```

Der Compiler vergleicht den Typ des Parameters mit dem des Prototypen in der Headerdatei `stdio.h`und findet dort keine Übereinstimmung. Nun "weiß" er, dass der Anweisung ein falscher Parameter übergeben wurde und gibt eine Fehlermeldung aus.

Das Konzept der Prototypen wurde als erstes in C++ eingeführt und war in der ursprünglichen Sprachdefinition von Kernighan und Ritchie noch nicht vorhanden. Deshalb kam auch beispielsweise das "Hello World" Programm in der ersten Auflage von "The C Programming Language" ohne `include`Anweisung aus. Erst mit der Einführung des ANSI Standards wurden auch in C Prototypen eingeführt.

7.3 Inline-Funktionen

Neu im C99-Standard sind Inline-Funktionen. Sie werden definiert, indem das Schlüsselwort `inline`vorangestellt wird. Beispiel:

```
inline float zylinder_oberflaeche(float h, float r)
{
  float o;
  o = 2 * 3.141 * r * (r + h);
  return(o);
}
```

Eine Funktion, die als `inline`definiert ist, soll gemäß dem C-Standard so schnell wie möglich aufgerufen werden. Die genaue Umsetzung ist der Implementierung überlassen. Beispielsweise kann der Funktionsaufruf dadurch beschleunigt werden, dass die Funktion nicht mehr als eigenständiger Code vorliegt, sondern an der Stelle des Funktionsaufrufs eingefügt wird. Dadurch entfällt eine Sprunganweisung in die Funktion und wieder zurück. Allerdings muss der Compiler das Schlüsselwort `inline`nicht beachten, wenn der Compiler keinen Optimierungsbedarf feststellt. Viele Compiler ignorieren deshalb dieses Schlüsselwort vollständig und setzen auf Heuristiken, wann eine Funktion inline sein sollte.

7.4 Globale und lokale Variablen

Alle bisherigen Beispielprogramme verwendeten lokale Variablen. Sie wurden am Beginn einer Funktion deklariert und galten nur innerhalb dieser Funktion. Sobald die Funktion verlassen wird verliert sie ihre Gültigkeit. Eine Globale Variable dagegen wird außerhalb einer Funktion deklariert (in der Regel am Anfang des Programms) und behält bis zum Beenden des Programms ihre Gültigkeit und dementsprechend einen Wert.

```
#include <stdio.h>

int GLOBAL_A = 43;
int GLOBAL_B = 12;

void funktion1( );
```

```
void funktion2( );

int main( void )
{
    printf( "Beispiele für lokale und globale Variablen: \n\n" );
    funktion1( );
    funktion2( );
    return 0;
}

void funktion1( )
{
    int lokal_a = 18;
    int lokal_b = 65;
    printf( "\nGlobale Variable A: %i", GLOBAL_A );
    printf( "\nGlobale Variable B: %i", GLOBAL_B );
    printf( "\nLokale Variable a: %i", lokal_a );
    printf( "\nLokale Variable b: %i", lokal_b );
}

void funktion2( )
{
    int lokal_a = 45;
    int lokal_b = 32;
    printf( "\n\nGlobale Variable A: %i", GLOBAL_A );
    printf( "\nGlobale Variable B: %i", GLOBAL_B );
    printf( "\nLokale Variable a: %i", lokal_a );
    printf( "\nLokale Variable b: %i \n", lokal_b );
}
```

Die Variablen `GLOBAL_A`und `GLOBAL_B`sind zu Beginn des Programms und außerhalb der Funktion deklariert worden und gelten deshalb im ganzen Programm. Sie können innerhalb jeder Funktion benutzt werden. Lokale Variablen wie `lokal_a`und `lokal_b`dagegen gelten nur innerhalb der Funktion, in der sie deklariert wurden. Sie verlieren außerhalb dieser Funktion ihre Gültigkeit. Der Compiler erzeugt deshalb beim Aufruf der Variable `lokal_a`einen Fehler, da die Variable in `Funktion1`deklariert wurde.

Globale Variablen unterscheiden sich in einem weiteren Punkt von den lokalen Variablen: Sie werden automatisch mit dem Wert 0 initialisiert wenn ihnen kein Wert zugewiesen wird. Lokale Variablen dagegen erhalten immer den (zufälligen) Wert, der sich gerade an der vom Compiler reservierten Speicherstelle befindet (Speichermüll). Diesen Umstand macht das folgende Programm deutlich:

```
#include <stdio.h>

int ZAHL_GLOBAL;

int main( void )
{
    int zahl_lokal;
    printf( "Lokale Variable: %i", zahl_lokal );
    printf( "\nGlobale Variable: %i \n", ZAHL_GLOBAL );
    return 0;
}
```

Das Ergebnis:

```
Lokale Variable: 296
Globale Variable: 0
```

7.4.1 Verdeckung

Sind zwei Variablen mit demselben Namen als globale und lokale Variable definiert, wird immer die lokale Variable bevorzugt. Das nächste Beispiel zeigt eine solche ”Doppeldeklaration”:

```
#include <stdio.h>

int zahl = 5;
void func( );

int main( void )
{
    int zahl = 3;
    printf( "Ist die Zahl %i als eine lokale oder globale Variable deklariert?",
 zahl );
    func( );
    return 0;
}

void func( )
{
    printf( "\nGlobale Variable: %i \n", zahl );
}
```

Neben der globalen Variable `zahl`wird in der Hauptfunktion `main`eine weitere Variable mit dem Namen `zahl`deklariert. Die globale Variable wird durch die lokale *verdeckt*. Da nun zwei Variablen mit demselben Namen existieren, gibt die `printf`Anweisung die lokale Variable mit dem Wert 3 aus. Die Funktion `func`soll lediglich verdeutlichen, dass die globale Variable `zahl`nicht von der lokalen Variablendeklaration gelöscht oder überschrieben wurde.

Man sollte niemals Variablen durch andere verdecken, da dies das intuitive Verständnis behindert und ein Zugriff auf die globale Variable im Wirkungsbereich der lokalen Variable nicht möglich ist. Gute Compiler können so eingestellt werden, dass sie eine Warnung ausgeben, wenn Variablen verdeckt werden.

Ein weiteres (gültiges) Beispiel für Verdeckung ist

```
#include <stdio.h>


int main( void )
{
    int i;
    for( i = 0; i<10; ++i )
    {
        int i;
        for( i = 0; i<10; ++i )
        {
            int i;
            for( i = 0; i<10; ++i )
            {
                printf( "i = %d \n", i );
            }
        }
    }
    return 0;
}
```

Hier werden 3 verschiedene Variablen mit dem Namen `i`angelegt, aber nur das innerste `i`ist für das `printf`von Belang. Dieses Beispiel ist intuitiv schwer verständlich und sollte auch nur ein Negativbeispiel sein.

7.5 exit()

Mit der Bibliotheksfunktion `exit()`kann ein Programm an einer beliebigen Stelle beendet werden. In Klammern muss ein Wert übergeben werden, der an die Umgebung - also in der Regel das Betriebssystem - zurückgegeben wird. Der Wert 0 wird dafür verwendet, um zu signalisieren, dass das Programm korrekt beendet wurde. Ist der Wert ungleich 0, so ist es implementierungsabhängig, welche Bedeutung der Rückgabewert hat. Beispiel:

```
exit(2);
```

Beendet das Programm und gibt den Wert 2 an das Betriebssystem zurück. Alternativ dazu können auch die Makros `EXIT_SUCCESS`und `EXIT_FAILURE`verwendet werden, um eine erfolgreiche bzw. fehlerhafte Beendigung des Programms zurückzuliefern.

Anmerkung: Unter DOS kann dieser Rückgabewert beispielsweise mittels IF ERRORLEVEL in einer Batchdatei ausgewertet werden, unter Unix/Linux enthält die spezielle Variable $? den Rückgabewert des letzten aufgerufenen Programms. Andere Betriebssysteme haben ähnliche Möglichkeiten; damit sind eigene Miniprogramme möglich, welche bestimmte Begrenzungen (von z.B. Batch- oder anderen Scriptsprachen) umgehen können. Sie sollten daher immer Fehlercodes verwenden, um das Ergebnis auch anderen Programmen zugänglich zu machen.

8 Eigene Header

Eigene Module mit den entsprechenden eigenen Headern sind sinnvoll, um ein Programm in Teilmodule zu zerlegen oder bei Funktionen und Konstanten, die in mehreren Programmen verwendet werden sollen. Eine Headerdatei – kurz: Header – hat die Form myheader.h. Sie enthält die Funktionsprototypen und Definitionen, die mit diesem Header in das Programm eingefügt werden.

```
#ifndef MYHEADER_H
#define MYHEADER_H

#define PI (3.1416)

extern int meineVariable;

int meineFunktion1(int);
int meineFunktion2(char);

#endif /* MYHEADER_H */
```

Anmerkung: Die Präprozessor-Direktiven `#ifndef`, `#define`und `#endif`werden detailliert im Kapitel Präprozessor[1] erklärt.

In der ersten Zeile dieses kleinen Beispiels überprüft der Präprozessor, ob im Kontext des Programms das Makro `MYHEADER_H`schon definiert ist. Wenn ja, ist auch der Header dem Programm schon bekannt und wird nicht weiter abgearbeitet. Dies ist nötig, weil es auch vorkommen kann, dass ein Header die Funktionalität eines andern braucht und diesen mit einbindet, oder weil im Header Definitionen wie Typdefinitionen mit typedef stehen, die bei Mehrfach-Includes zu Compilerfehlern führen würden.

Wenn das Makro `MYHEADER_H`dem Präprozessor noch nicht bekannt ist, dann beginnt er ab der zweiten Zeile mit der Abarbeitung der Direktiven im if-Block. Die zweite Zeile gibt dem Präprozessor die Anweisung, das Makro `MYHEADER_H`zu definieren. Damit wird gemerkt, dass dieser Header schon eingebunden wurde. Dieser Makroname ist frei wählbar, muss im Projekt jedoch eindeutig sein. Es hat sich die Konvention etabliert, den Namen dieses Makros zur Verbesserung der Lesbarkeit an den Dateinamen des Headers anzulehnen und ihn als `MYHEADER_H`oder `__MYHEADER_H__`zu wählen. Dann wird der Code von Zeile 3 bis 10 in die Quelldatei, welche die #include-Direktive enthält, eingefügt. Zeile 11 kommt bei der Headerdatei immer am Ende und teilt dem Präprozessor das Ende des if-Zweigs (siehe Kapitel Präprozessor[2]) mit.

Variablen allgemein verfügbar zu machen stellt ein besonderes Problem dar, das besonders für Anfänger schwer verständlich ist. Grundsätzlich sollte man den Variablen in Header-

1 Kapitel 17 auf Seite 133
2 Kapitel 17 auf Seite 133

Dateien das Schlüsselwort *extern*voranstellen. Damit erklärt man dem Compiler, dass es die Variable *meineVariable*gibt, diese jedoch an anderer Stelle definiert ist.

Würde eine Variable in einer Header-Datei definiert werden, würde für jede C-Datei, die die Header-Datei einbindet, eine eigene Variable mit eigenem Speicher erstellt. Jede C-Datei hätte also ein eigenes Exemplar, ohne dass sich deren Bearbeitung auf die Variablen, die die anderen C-Dateien kennen, auswirkt. Eine Verwendung solcher Variablen sollte vermieden werden, da sie vor allem in der hardwarenahen Programmierung der Ressourcenschonung dient. Stattdessen sollte man Funktionen der Art `int getMeineVariable()`benutzen.

Nachdem die Headerdatei geschrieben wurde, ist es noch nötig, eine C-Datei myheader.c zu schreiben. In dieser Datei werden die in den Headerzeilen 8 und 9 deklarierten Funktionen implementiert. Damit der Compiler weiß, dass diese Datei die Funktionalität des Headers ausprägt, wird als erstes der Header inkludiert; danach werden einfach wie gewohnt die Funktionen geschrieben.

```
#include "myheader.h"

int meineVariable = 0;

int meineFunktion1 (int i)
{
  return (i+1);
}

int meineFunktion2 (char c)
{
  if (c == 'A')
    return 1;
  return 0;
}
```

Die Datei myheader.c wird jetzt kompiliert und eine so genannte Objektdatei erzeugt. Diese hat typischerweise die Form myheader.obj oder myheader.o. Zuletzt muss dem eigentlichen Programm die Funktionalität des Headers bekannt gemacht werden, wie es durch ein `#include "myheader.h"`geschieht, und dem Linker muss beim Erstellen des Programms gesagt werden, dass er die Objektdatei myheader.obj bzw. myheader.o mit einbinden soll.

Damit der im Header verwiesenen Variable auch eine real existierende gegenübersteht, muss in myheader.c eine Variable vom selben Typ und mit demselben Namen definiert werden.

9 Zeiger

Eine Variable wurde bisher immer direkt über ihren Namen angesprochen. Um zwei Zahlen zu addieren, wurde beispielsweise der Wert einem Variablennamen zugewiesen:

```
summe = 5 + 7;
```

Eine Variable wird intern im Rechner allerdings immer über eine Adresse angesprochen (außer die Variable befindet sich bereits in einem Prozessorregister). Alle Speicherzellen innerhalb des Arbeitsspeichers erhalten eine eindeutige Adresse. Immer wenn der Prozessor einen Wert aus dem RAM liest oder schreibt, schickt er diese über den Systembus an den Arbeitsspeicher.

Eine Variable kann in C auch direkt über die Adresse angesprochen werden. Eine Adresse liefert der &-Operator (auch als Adressoperator[1] bezeichnet). Diesen Adressoperator kennen Sie bereits von der `scanf`-Anweisung:

```
scanf("%i", &a);
```

Wo diese Variable abgelegt wurde, lässt sich mit einer `printf`Anweisung herausfinden:

```
printf("%p\n", &a);
```

Der Wert kann sich je nach Betriebssystem, Plattform und sogar von Aufruf zu Aufruf unterscheiden. Der Platzhalter `%p`steht für das Wort *Zeiger*(engl.: *pointer*).

Eine Zeigervariable dient dazu, ein Objekt (z.B. eine Variable) über ihre Adresse anzusprechen. Im Gegensatz zu einer "normalen" Variable, erhält eine Zeigervariable keinen Wert, sondern eine Adresse.

9.1 Beispiel

Im folgenden Programm wird die Zeigervariable a deklariert:

```
#include <stdio.h>

int main(void)
{
  int *a, b;

  b = 17;
  a = &b;
  printf("Inhalt der Variablen b:    %i\n", b);
  printf("Inhalt des Speichers der Adresse auf die a zeigt:    %i\n", *a);
```

1 Kapitel 22.6.3 auf Seite 164

```
  printf("Adresse der Variablen b:   %p\n", &b);
  printf("Adresse auf die die Zeigervariable a verweist:   %p\n", (void *)a);
  /* Aber */
  printf("Adresse der Zeigervariable a: %p\n", &a);
  return 0;
}
```

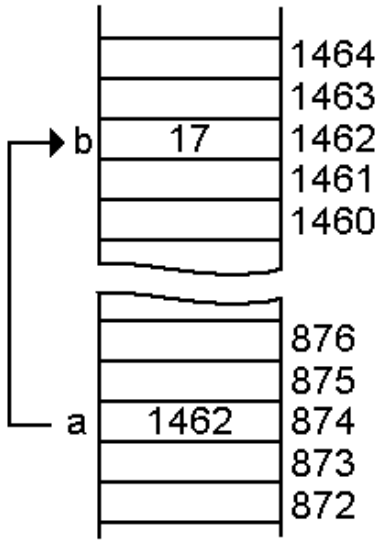


Abb. 3 Abb. 1 - Das (vereinfachte) Schema zeigt wie das Beispielprogramm arbeitet. Der Zeiger `a`zeigt auf die Variable `b`. Die Speicherstelle des Zeigers `a`besitzt lediglich die Adresse von `b`(im Beispiel 1462). Hinweis: Die Adressen für die Speicherzellen sind erfunden und dienen lediglich der besseren Illustration.

In Zeile 5 wird die Zeigervariable `a`deklariert. Dabei wird aber kein eigener Speicherbereich für die Variable `a`selbst bereitgestellt, sondern **ein Speicherbereich für die Adresse**! Außerdem wird eine Integervariable des Typs `int`deklariert. Bitte beachten Sie, dass die Anweisung

```
int* a, b;
```

einen Zeiger auf die Integer-Variable `a`und **nicht**die Integer-Variable `b`deklariert (b ist also kein Zeiger!). Deswegen sollte man sich angewöhnen, den Stern zum Variablennamen und nicht zum Datentyp zu schreiben:

```
int *a, b;
```

Diese Schreibweise verringert die Verwechslungsgefahr deutlich.

Nach der Deklaration hat die Zeigervariable `a`einen nicht definierten Inhalt. Die Anweisung `a=&b`in Zeile 8 weist `a`deshalb eine neue Adresse zu. Damit zeigt die Variable `a`nun auf die Variable `b`.

Die `printf`-Anweisung gibt den Wert der Variable aus, auf die der Zeiger verweist. Da ihr die Adresse von `b`zugewiesen wurde, wird die Zahl 17 ausgegeben.

Ob Sie auf den Inhalt der Adresse auf den die Zeigervariable verweist oder auf die Adresse auf den die Zeigervariable verweist zugreifen, hängt vom * - Operator ab:

- *a = greift auf den Inhalt der Zeigervariable zu. Der *-Operator wird auch als Inhalts- oder Dereferenzierungs-Operator [2] bezeichnet.
- a = greift auf die Adresse, auf die die Zeigervariable verweist, zu.

Will man aber die Adresse der Zeigervariable selbst haben, so muss man den & Operator wählen. Also so: &a.

Ein Zeiger darf nur auf eine Variable verweisen, die denselben Datentyp hat. Ein Zeiger vom Typ `int`kann also nicht auf eine Variable mit dem Typ `float`verweisen. Den Grund hierfür werden Sie im nächsten Kapitel kennen lernen. Nur so viel vorab: Der Variablentyp hat nichts mit der Breite der Adresse zu tun. Diese ist systemabhängig immer gleich. Bei einer 16 Bit CPU ist die Adresse 2 Byte, bei einer 32 Bit CPU 4 Byte und bei einer 64 Bit CPU 8 Byte breit - unabhängig davon, ob die Zeigervariable als char, int, float oder double deklariert wurde.

9.2 Zeigerarithmetik

Es ist möglich, Zeiger zu erhöhen und damit einen anderen Speicherbereich anzusprechen, z. B.:

```
#include <stdio.h>

int main()
{
  int x = 5;
  int *i = &x;
  printf("Speicheradresse %p enthält %i\n", (void *)i, *i);
  i++; // nächste Adresse lesen
  printf("Speicheradresse %p enthält %i\n", (void *)i, *i);
  return 0;
}
```

`i++`erhöht hier **nicht**den *Inhalt*`(*i)`, sondern die *Adresse*des Zeigers `(i)`. Man sieht aufgrund der Ausgabe auch leicht, wie groß ein int auf dem System ist, auf dem das Programm kompiliert wurde. Im folgenden handelt es sich um ein 32-bit-System (Differenz der beiden Speicheradressen 4 Byte = 32 Bit):

```
Speicheradresse 134524936 enthält 5
Speicheradresse 134524940 enthält 0
```

2 Kapitel 22.6.3 auf Seite 164

Um nun den Wert im Speicher, nicht den Zeiger, zu erhöhen, wird `*i++`nichts nützen. Das ist so, weil der Dereferenzierungsoperator `*`die niedrigere Priorität[3] hat als das Postinkrement (`i++`). Um den beabsichtigten Effekt zu erzielen, schreibt man `(*i)++`, oder auch `++*i`. Im Zweifelsfall und auch um die Les- und Wartbarkeit zu erhöhen sind Klammern eine gute Wahl.

9.3 Zeiger auf Funktionen

Zeiger können nicht nur auf Variablen, sondern auch auf Funktionen verweisen, da Funktionen nichts anderes als Code im Speicher sind. Ein Zeiger auf eine Funktion erhält also die Adresse des Codes.

Mit dem folgenden Ausdruck wird ein Zeiger auf eine Funktion definiert:

```
int (*f) (float);
```

Diese Schreibweise erscheint zunächst etwas ungewöhnlich. Bei genauem Hinsehen gibt es aber nur einen Unterschied zwischen einer normalen Funktionsdefinition und der Zeigerschreibweise: Anstelle des Namens der Funktion tritt der Zeiger. Der Variablentyp int ist der Rückgabetyp und float der an die Funktion übergebene Parameter. Die Klammer um den Zeiger darf nicht entfernt werden, da der Klammeroperator () eine höhere Priorität als der Dereferenzierungsoperator * hat.

Wie bei einer Zeigervariable kann ein Zeiger auf eine Funktion nur eine Adresse aufnehmen. Wir müssen dem Zeiger also noch eine Adresse zuweisen:

```
int (*f) (float);
int func(float);
f = func;
```

Die Schreibweise (f = func) ist gleich mit (f = &func), da die Adresse der Funktion im Funktionsnamen steht. Der Lesbarkeit halber sollte man nicht auf den Adressoperator(&) verzichten.

Die Funktion können wir über den Zeiger nun wie gewohnt aufrufen:

```
(*f)(35.925);
```

oder

```
f(35.925);
```

Hier ein vollständiges Beispielprogramm:

```
#include <stdio.h>

int zfunc( )
{
    int var1 = 2009;
    int var2 = 6;
```

3 Kapitel 22.6.10 auf Seite 172

```
    int var3 = 8;

    printf( "Das heutige Datum lautet: %d.%d.%d\n", var3, var2, var1 );
    return 0;
}

int main( void )
{
    int var1 = 2010;
    int var2 = 7;
    int var3 = 9;
    int ( *f )( );

    f = &zfunc;

    printf( "Ich freue mich schon auf den %d.%d.%d\n", var3, var2, var1 );
    f( );
    printf( "Die Adresse der Funktion im RAM lautet: %p\n", (void *)f );
    return 0;
}
```

9.4 void-Zeiger

Der void-Zeiger ist zu jedem Datentyp kompatibel (Achtung, anders als in C++[4]). Man spricht hierbei auch von einem untypisierten oder generischen Zeiger. Das geht so weit, dass man einen void Zeiger in jeden anderen Zeiger umwandeln kann, und zurück, ohne dass die Repräsentation des Zeigers Eigenschaften verliert. Ein solcher Zeiger wird beispielsweise bei der Bibliotheksfunktion malloc[5] benutzt. Diese Funktion wird verwendet um eine bestimmte Menge an Speicher bereitzustellen, zurückgegeben wird die Anfangsadresse des allozierten Bereichs. Danach kann der Programmierer Daten beliebigen Typs dorthin schreiben und lesen. Daher ist Pointer-Typisierung irrelevant. Der Prototyp von malloc[6] ist also folgender:

```
void *malloc(size_t size);
```

Der Rückgabetyp `void*`ist hier notwendig, da ja nicht bekannt ist, welcher Zeigertyp (`char*`, `int*`usw.) zurückgegeben werden soll. Vielmehr ist es möglich, den Typ `void*`in jeden Zeigertyp zu "*casten*" (umzuwandeln, vgl. type-cast = Typumwandlung[7]).

Der einzige Unterschied zu einem typisierten ("normalen") Zeiger ist, dass die Zeigerarithmetik schwer zu bewältigen ist, da dem Compiler der Speicherplatzverbrauch pro Variable nicht bekannt ist (wir werden darauf im nächsten Kapitel[8] noch zu sprechen kommen) und man in diesem Fall sich selber darum kümmern muss, dass der void Pointer auf der richtigen Adresse zum Liegen kommt. Zum Beispiel mit Hilfe des *sizeof*Operator.

```
int *intP;
void *voidP;
voidP = intP;         /* beide zeigen jetzt auf das gleiche Element */
intP++;               /* zeigt nun auf das nächste Element */
voidP += sizeof(int); /* zeigt jetzt auch auf das nächste int Element */
```

4 http://de.wikibooks.org/wiki/C%2B%2B-Programmierung
5 Kapitel 14 auf Seite 125
6 Kapitel 14 auf Seite 125
7 Kapitel 13 auf Seite 123
8 Kapitel 10.11 auf Seite 99

9.5 Unterschied zwischen Call by Value und Call by Reference

Eine Funktion dient dazu, eine bestimmte Aufgabe zu erfüllen. Dazu können ihr Variablen übergeben werden oder sie kann einen Wert zurückgeben. Der Compiler übergibt diese Variable aber nicht direkt der Funktion, sondern fertigt eine Kopie davon an. Diese Art der Übergabe von Variablen wird als *Call by Value*bezeichnet.

Da nur eine Kopie angefertigt wird, gelten die übergebenen Werte nur innerhalb der Funktion selbst. Sobald die Funktion wieder verlassen wird, gehen alle diese Werte verloren. Das folgende Beispiel verdeutlicht dies:

```
#include <stdio.h>

void func(int wert)
{
  wert += 5;
  printf("%i\n", wert);
}

int main()
{
  int zahl = 10;
  printf("%i\n", zahl);
  func(zahl);
  printf("%i\n", zahl);
  return 0;
}
```

Das Programm erzeugt nach der Kompilierung die folgende Ausgabe auf dem Bildschirm:

```
10
15
10
```

Dies kommt dadurch zustande, dass die Funktion `func`nur eine Kopie der Variable `wert`erhält. Zu dieser Kopie addiert dann die Funktion `func`die Zahl 5. Nach dem Verlassen der Funktion geht der Inhalt der Variable `wert`verloren. Die letzte `printf`Anweisung in `main`gibt deshalb wieder die Zahl 10 aus.

Eine Lösung wurde bereits im Kapitel Funktionen[9] angesprochen: Die Rückgabe über die Anweisung `return`. Diese hat allerdings den Nachteil, dass jeweils nur ein Wert zurückgegeben werden kann.

Ein gutes Beispiel dafür ist die `swap()`Funktion. Sie soll dazu dienen, zwei Variablen zu vertauschen. Die Funktion müsste in etwa folgendermaßen aussehen:

```
void swap(int x, int y)
{
  int tmp;
  tmp = x;
  x = y;
```

9 Kapitel 7 auf Seite 65

```
   y = tmp;
 }
```

Die Funktion ist zwar prinzipiell richtig, kann aber das Ergebnis nicht an die Hauptfunktion zurückgeben, da `swap`nur mit Kopien der Variablen `x`und `y`arbeitet.

Das Problem lässt sich lösen, indem nicht die Variable direkt, sondern - Sie ahnen es sicher schon - ein Zeiger auf die Variable der Funktion übergeben wird. Das richtige Programm sieht dann folgendermaßen aus:

```
#include <stdio.h>

void swap(int *x, int *y)
{
  int tmp;
  tmp = *x;
  *x = *y;
  *y = tmp;
}

int main()
{
  int x = 2, y = 5;
  printf("Variable x: %i, Variable y: %i\n", x, y);
  swap(&x, &y);
  printf("Variable x: %i, Variable y: %i\n", x, y);
  return 0;
}
```

In diesem Fall ist das Ergebnis richtig:

```
  Variable x: 2, Variable y: 5
  Variable x: 5, Variable y: 2
```

Das Programm ist nun richtig, da die Funktion `swap`nun nicht mit den Kopien der Variable x und y arbeitet, sondern mit den Originalen. In vielen Büchern wird ein solcher Aufruf auch als *Call By Reference*bezeichnet. Diese Bezeichnung ist aber nicht unproblematisch. Tatsächlich liegt auch hier ein *Call By Value*vor, allerdings wird nicht der Wert der Variablen sondern deren Adresse übergeben. C++ und auch einige andere Sprachen unterstützen ein echtes *Call By Reference*, C hingegen nicht.

9.6 Verwendung

Sie stellen sich nun möglicherweise die Frage, welchen Nutzen man aus Zeigern zieht. Es macht den Anschein, dass wir, abgesehen vom Aufruf einer Funktion mit Call by Reference, bisher ganz gut ohne Zeiger auskamen. Andere Programmiersprachen scheinen sogar ganz auf Zeiger verzichten zu können. Dies ist aber ein Trugschluss: Häufig sind Zeiger nur gut versteckt, so dass nicht auf den ersten Blick erkennbar ist, dass sie verwendet werden. Beispielsweise arbeitet der Rechner bei Zeichenketten intern mit Zeigern, wie wir noch sehen werden. Auch das Kopieren, Durchsuchen oder Verändern von Datenfeldern ist ohne Zeiger nicht möglich.

Es gibt Anwendungsgebiete, die ohne Zeiger überhaupt nicht auskommen: Ein Beispiel hierfür sind Datenstrukturen wie beispielsweise verkettete Listen, die wir später noch kurz kennen lernen. Bei verketteten Listen werden die Daten in einem sogenannten Knoten gespeichert. Diese Knoten sind untereinander jeweils mit Zeigern verbunden. Dies hat den Vorteil, dass die Anzahl der Knoten und damit die Anzahl der zu speichernden Elemente dynamisch wachsen kann. Soll ein neues Element in die Liste eingefügt werden, so wird einfach ein neuer Knoten erzeugt und durch einen Zeiger mit der restlichen verketteten Liste verbunden. Es wäre zwar möglich, auch für verkettete Listen eine zeigerlose Variante zu implementieren, dadurch würde aber viel an Flexibilität verloren gehen. Auch bei vielen anderen Datenstrukturen und Algorithmen kommt man ohne Zeiger nicht aus. Einige Algorithmen lassen sich darüber hinaus mithilfe von Zeigern auch effizienter implementieren, so dass deren Ausführungszeit schneller als die Implementierung des selben Algorithmus ohne Zeiger ist.

en:C Programming/Pointers and arrays[10] it:C/Vettori e puntatori/Interscambiabilità tra puntatori e vettori[11] pl:C/Wskaźniki[12]

10 http://en.wikibooks.org/wiki/C%20Programming%2FPointers%20and%20arrays

11 http://it.wikibooks.org/wiki/C%2FVettori%20e%20puntatori%2FInterscambiabilit%C3%A0%20tra%20puntatori%20e%20vettori

12 http://pl.wikibooks.org/wiki/C%2FWska%C5%BAniki

10 Arrays

10.1 Eindimensionale Arrays

Nehmen Sie einmal rein fiktiv an, Sie wollten ein Programm für Ihre kleine Firma schreiben, das die Summe sowie den höchsten und den niedrigsten Umsatz der Umsätze einer Woche ermittelt. Es wäre natürlich sehr ungeschickt, wenn Sie die Variable `umsatz1`bis `umsatz7`deklarieren müssten. Noch umständlicher wäre die Addition der Werte und das Ermitteln des höchsten bzw. niedrigsten Umsatzes.

Für die Lösung des Problems werden stattdessen Arrays (auch als Felder oder Vektoren bezeichnet) benutzt. Arrays unterscheiden sich von normalen Variablen lediglich darin, dass sie einen Index besitzen. Statt `umsatz1`bis `umsatz7`zu deklarieren, reicht die einmalige Deklaration von

```
float umsatz[7];


Visuelle Darstellung:
Index: | [0] | [1] | [2] | [3] | [4] | [5] | [6] | ...
Werte: | [ ] | [ ] | [ ] | [ ] | [ ] | [ ] | [ ] | ...
```

aus. Damit deklarieren Sie in einem Rutsch die Variablen `umsatz[0]`bis `umsatz[6]`. Beachten Sie unbedingt, dass auf ein Array immer mit dem Index 0 beginnend zugegriffen wird! Beispielsweise wird der fünfte Wert mit dem Index 4 (`umsatz[4]`) angesprochen! Dies wird nicht nur von Anfängern gerne vergessen und führt auch bei erfahreneren Programmierern häufig zu „Um-eins-daneben-Fehlern[1]“.

Die Addition der Werte erfolgt in einer Schleife. Der Index muss dafür in jedem Durchlauf erhöht werden. In dieser Schleife testen wir gleichzeitig jeweils beim Durchlauf, ob wir einen niedrigeren oder einen höheren Umsatz als den bisherigen Umsatz haben:

```
#include <stdio.h>

int main( void )
{
    float umsatz[7];
    float summe, hoechsterWert, niedrigsterWert;
    int i;

    for( i = 0; i < 7; i++ )
    {
        printf( "Bitte die Umsaetze der letzten Woche eingeben: \n" );
        scanf( "%f", &umsatz[i] );
    }
```

1 http://de.wikipedia.org/wiki/Off-by-one-Error

```
    summe = 0;
    hoechsterWert = umsatz[0];
    niedrigsterWert = umsatz[0];

    for( i = 0; i < 7; i++ )
    {
        summe += umsatz[ i ];
        if( hoechsterWert < umsatz[i] )
            hoechsterWert = umsatz[i];
        //
        if( niedrigsterWert > umsatz[i] )
            niedrigsterWert = umsatz[i];
    }

    printf( "Gesamter Wochengewinn: %f \n", summe );
    printf( "Hoechster Umsatz: %f \n", hoechsterWert );
    printf( "Niedrigster Umsatz: %f \n", niedrigsterWert );
    return 0;
}
```

ACHTUNG: Bei einer Zuweisung von Arrays wird nicht geprüft, ob eine Feldüberschreitung vorliegt. So führt beispielsweise

```
umsatz[10] = 5.0;
```

nicht zu einer Fehlermeldung, obwohl das Array nur 7 Elemente besitzt. Der Compiler gibt weder eine Fehlermeldung noch eine Warnung aus! Der Programmierer ist selbst dafür verantwortlich, dass die Grenzen des Arrays nicht überschritten werden. Ein Zugriff auf ein nicht vorhandenes Arrayelement kann zum Absturz des Programms oder anderen unvorhergesehenen Ereignissen führen! Des Weiteren kann dies ein sehr hohes Sicherheitsrisiko darstellen. Denn ein Angreifer kann dann über das Array eigene Befehle in den Arbeitsspeicher schreiben und vom Programm ausführen lassen. (Siehe Bufferoverflow[2])

10.2 Mehrdimensionale Arrays

Ein Array kann auch aus mehreren Dimensionen bestehen. Das heißt, es wird wie eine Matrix dargestellt. Im Folgenden wird beispielsweise ein Array mit zwei Dimensionen definiert:

2 http://de.wikipedia.org/wiki/Puffer%C3%BCberlauf

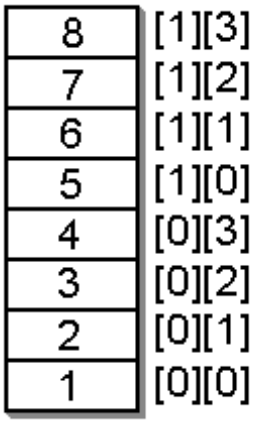
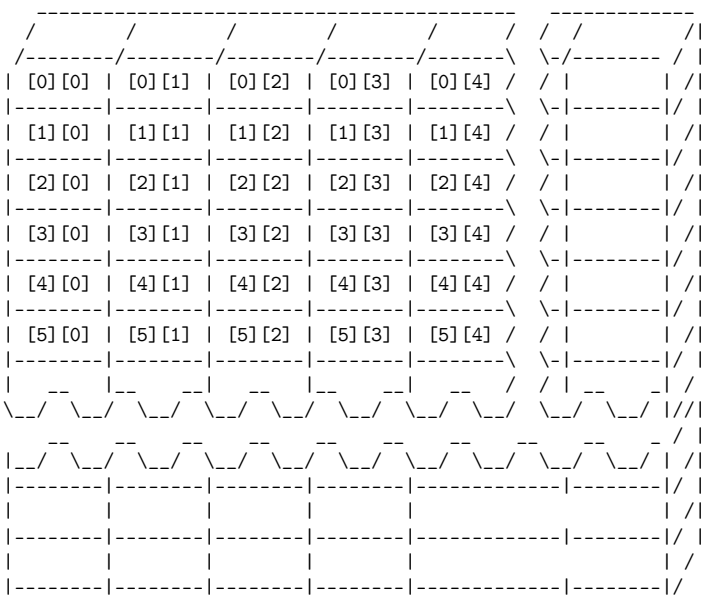


Abb. 4 Abb 1. zweidimensionales Array im Speicher

```
int vararray[6][5]

Visuelle Darstellung:
```

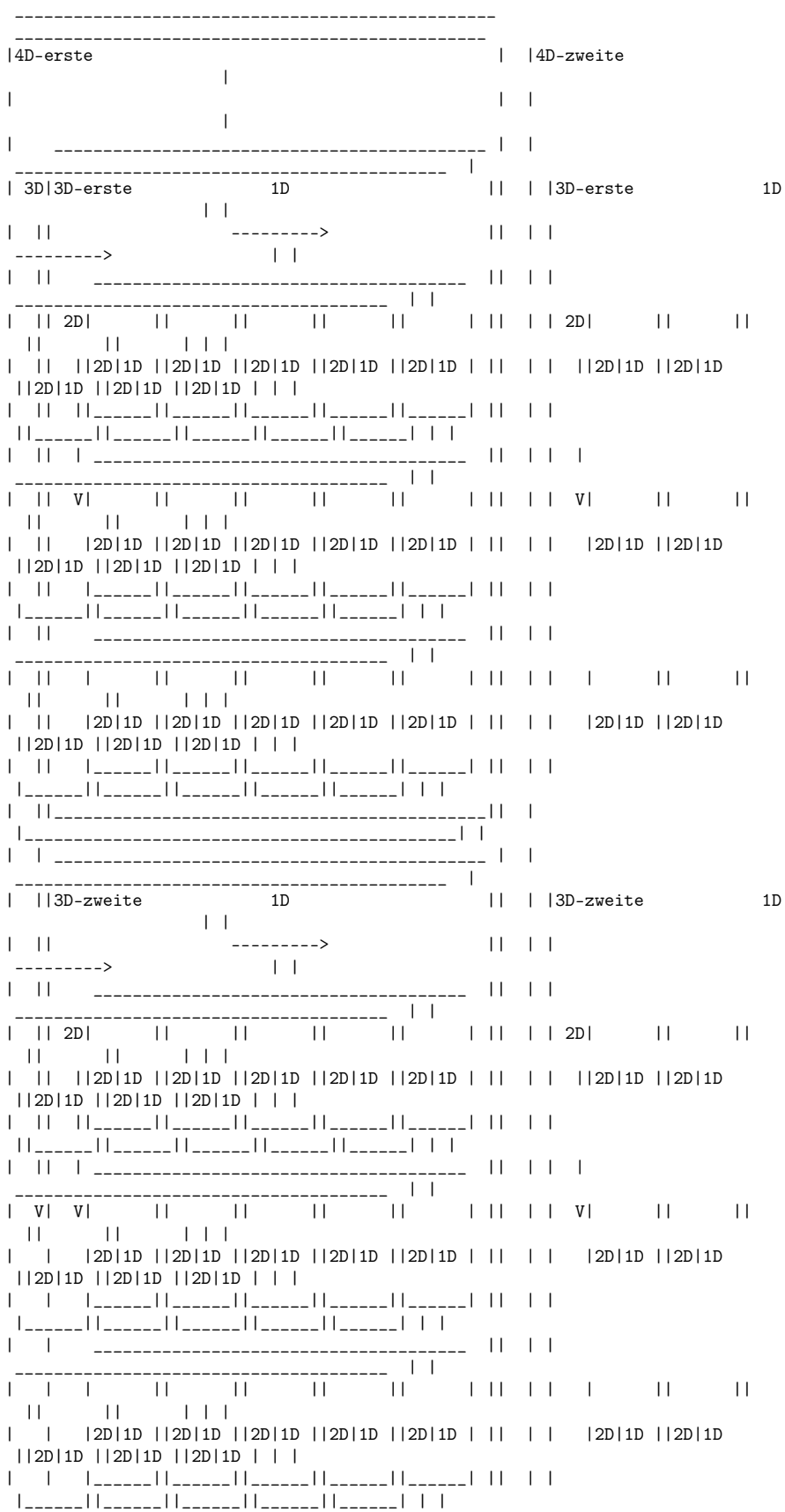


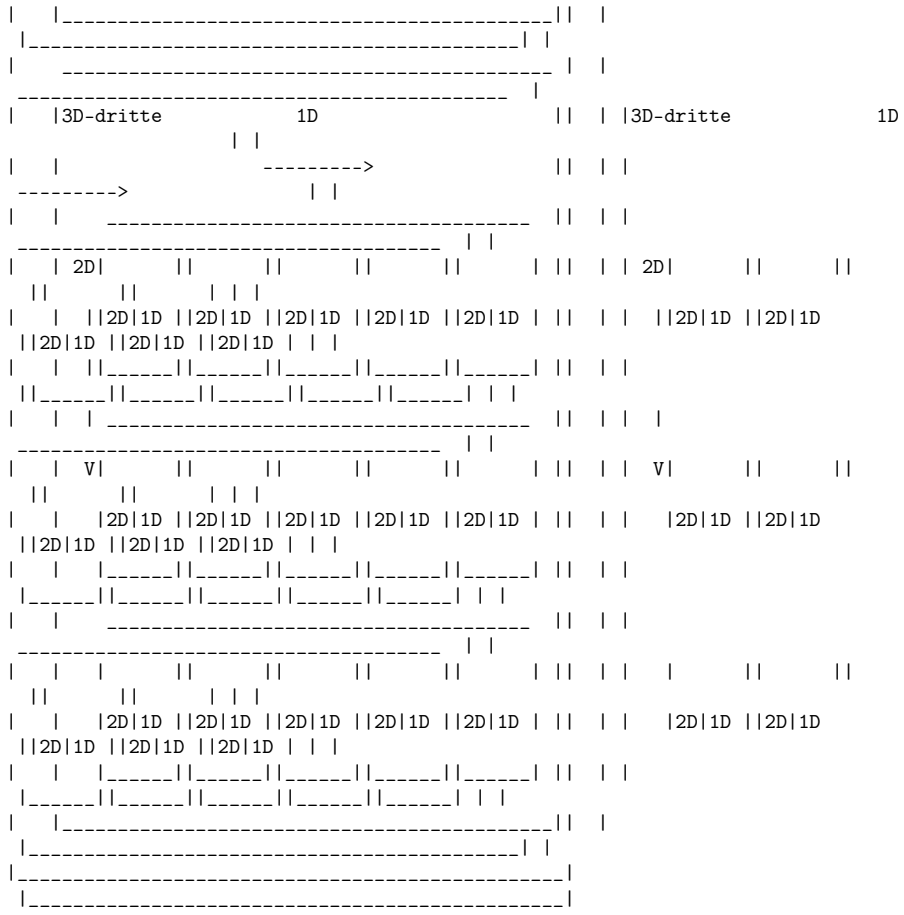
Wie aus der Abbildung 1 ersichtlich, entspricht das mehrdimensionale Array im Speicher im Prinzip einem eindimensionalen Array. Dies muss nicht verwundern, da der Speicher ja selbst eindimensional aufgebaut ist.

Ein mehrdimensionales Array wird aber dennoch häufig verwendet, etwa wenn es darum geht, eine Tabelle, Matrix oder Raumkoordinaten zu speichern.

Mehrdimensionales Array genauer betrachtet

```
int Ary[2][3][3][5];                              4D
                                                ------------------>
```


4D-erste
4D-zweite
3D|3D-erste 1D
3D-erste 1D
2D|1D
3D-zweite 1D
3D-zweite 1D



Der erste Index des Arrays steht für die vierte Dimension, der zweite Index für die dritte Dimension, der dritte Index für die zweite Dimension und der letzte Index für die erste Dimension. Dies soll veranschaulichen, wie man sich ein mehrdimensionales Array vorstellen muss.

Veranschaulichung

Weil die Vorstellung von Objekten als mehrdimensionale Arrays abseits von 3 Dimensionen (Würfel) schwierig ist, sollte man sich Arrays lieber als ein doppelte Fortschrittsbalken (wie bei einem Brennprogramm oft üblich) oder als Maßeinheit (z. B. Längenangaben) vorstellen. Um es an einem dieser genannten Beispiele zu veranschaulichen:

Man stellt sich einen Millimeter als erstes Array-Element (Feld) vor.

```
1 Feld = 1 mm

int Array[10];
#10 mm = 1 cm
#Array[Eine-Dimension (10 Felder)] = 1 cm
```

Natürlich könnte man mehr Felder für die erste Dimension verwenden, doch sollte man es zu Gunsten der Übersichtlichkeit nicht übertreiben.

```
int Array[10][10];
```

```
#10 mm x 10 = 1 dm
#Array[Zwei Dimensionen (Zehn Zeilen (eine Zeile mit je 10 Feldern)] = 1 dm
```

Die Anzahl der weiteren Feldblöcke (oder der gesamten Felder) wird durch die angegebene Zeilenanzahl bestimmt.

```
int Array[10][10][10]
#10 mm x 10 x 10 = 1 m
#Array[Drei-Dimensionen (Zehn mal _2D-Blöcke_ (die mit je 10 Feld-Blöcken, die
 wiederum mit je 10 Feldern)) ] = 1 m
```

Insgesamt enthält dieses Array somit 1000 Felder, in denen man genau so viele Werte speichern könnte wie Felder vorhanden. Die Dimensionen verlaufen von der kleinsten (1D) außen rechts zur größten (hier 3D) nach außen links.

Ab der dritten Dimension folgt es immer dem gleichem Muster.

Hier noch ein Beispielprogramm zum Verständnis:

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

#define WARY1  10
#define WARY2  10
#define WARY3  10


int main( void )
{
    srand( time( 0 ) );

    int a, b, c;
    int ZAry[WARY1][WARY2][WARY3];
    //
    for( a = 0; a < WARY1; ++a )
    {
        for( b = 0; b < WARY2; ++b )
        {
            for( c = 0; c < WARY3; ++c )
            {
                ZAry[a][b][c] = rand( );
            }
        }
    }
    //
    for( a = 0; a < WARY1; ++a )
    {
        for( b = 0; b < WARY2; ++b )
        {
            for( c = 0; c < WARY3; ++c )
            {
                printf( "Inhalt von Z-Ary[%d][%d][%d] ", a, b, c );
                printf( "ist: %d \n", ZAry[a][b][c] );
            }
            printf( "Weiter mit Eingabetaste && Verlassen mit STRG-C. \n" );
            getchar( );
        }
    }
    //
    return 0;
}
```

10.3 Arrays initialisieren

Es gibt zwei Schreibstile für die Initialisierung eines Arrays. Entweder die Werte gruppiert untereinander schreiben:

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

oder alles hintereinander schreiben:

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

Grundsätzlich ist es ratsam, ein Array immer zu initialisieren, damit man beim späterem Ausführen des Programms nicht durch unerwartete Ergebnisse überrascht wird. Denn ohne eine Initialisierung weiß man nie, welchen Wert die einzelnen Array-Elemente beinhalten. Doch sollte man aufpassen, nicht mehr zu initialisieren, als vorhanden ist, sonst liest das Programm nach dem Kompilieren und Ausführen einen willkürlichen negativen Wert.

Beispiel für eine Initialisierung, um sichere Werte zu haben:

```
int Ary[5] = { 0, 0, 0, 0, 0 };
```

10.4 Syntax der Initialisierung

Es gibt zwei Möglichkeiten ein Array zu initialisieren, entweder eine teilweise oder eine vollständige Initialisierung. Bei einer Initialisierung steht ein Zuweisungsoperator nach dem deklariertem Array, gefolgt von einer in geschweiften Klammern stehende Liste von Werten, die durch Komma getrennt werden. Diese Liste wird der Reihenfolge nach ab dem Index 0 den Array-Elementen zugewiesen.

10.5 Eindimensionales Array vollständig initialisiert

```
int Ary[5] = { 10, 20, 30, 40, 50 };

Index  | Inhalt
----------------
Ary[0] = 10
Ary[1] = 20
Ary[2] = 30
Ary[3] = 40
Ary[4] = 50
```

Fehlende Größenangabe bei vollständig initialisierten eindimensionalen Arrays

Wenn die Größe eines vollständig initialisierten eindimensionalen Arrays nicht angegeben wurde, erzeugt der Compiler ein Array, das gerade groß genug ist, um die Werte aus der Initialisierung aufzunehmen. Deshalb ist:

```
int Ary[5] = { 10, 20, 30, 40, 50 };
```

das gleiche wie:

```
int Ary[ ] = { 10, 20, 30, 40, 50 };
```

Ob man die Größe angibt oder weglässt ist jedem selbst überlassen, jedoch ist es zu empfehlen, sie anzugeben.

10.6 Eindimensionales Array teilweise initialisiert

```
int Ary[5] = { 10, 20, 30 };

Index  | Inhalt
----------------
Ary[0] = 10
Ary[1] = 20
Ary[2] = 30
Ary[3] = 0
Ary[4] = 0
```

Wie man hier in diesem Beispiel deutlich erkennt, werden nur die ersten drei Array-Elemente mit dem Index 0, 1 und 2 initialisiert. Somit sind diese Felder konstant mit den angegebenen Werten gefüllt und ändern sich ohne zutun nicht mehr. Hingegen sind die beiden letzten Array-Elemente, mit der Indexnummer 3 und 4, leer geblieben. Diese Felder werden bei solchen nur teilweise initialisierten Arrays vom Compiler mit dem Wert 0 gefüllt, um den Speicherplatz zu reservieren.

Fehlende Größenangabe bei teilweise initialisierten Eindimensionalen Arrays

Bei einem teilweise initialisierten eindimensionalen Array mit fehlender Größenangabe sieht es schon etwas anders aus. Dort führt eine fehlende Größenangabe dazu, dass die Größe des Arrays womöglich nicht ausreichend ist, weil nur so viele Array-Elemente vom Compiler erstellt wurden um die Werte aus der Liste aufzunehmen. Deshalb sollte man immer die Größe angeben!

10.7 Mehrdimensionales Array vollständig initialisiert

```
int Ary[4][5] = {
            { 10, 11, 12, 13, 14 },
            { 24, 25, 26, 27, 28 },
            { 30, 31, 32, 33, 34 },
            { 44, 45, 46, 47, 48 },
        }


Visuelle Darstellung:
---------------------
Index      | Inhalt
--------------------
Ary[0][0] = 10
Ary[0][1] = 11
```

```
Ary[0][2] = 12
Ary[0][3] = 13
Ary[0][4] = 14

Ary[1][0] = 24
Ary[1][1] = 25
Ary[1][2] = 26
Ary[1][3] = 27
Ary[1][4] = 28

Ary[2][0] = 30
Ary[2][1] = 31
Ary[2][2] = 32
Ary[2][3] = 33
Ary[2][4] = 34

Ary[3][0] = 44
Ary[3][1] = 45
Ary[3][2] = 46
Ary[3][3] = 47
Ary[3][4] = 48
```

Fehlende Größenangabe bei vollständig initialisierten mehrdimensionalen Arrays

Bei vollständig initalisierten mehrdimensionalen Array sieht es mit dem Weglassen der Größenangabe etwas anders aus als bei vollständig initialisierten eindimensionalen Arrays. Denn wenn ein Array mehr als eine Dimension besitzt, darf man nicht alle Größenangaben weg lassen. Grundsätzlich sollte man nie auf die Größenangaben verzichten. Notfalls ist es gestattet, die erste (und nur diese) weg zu lassen. Mit „erste“ ist immer die linke gemeint, die direkt an den Array Variablennamen angrenzt.

Wenn also eine Größenangabe (die erste) des Arrays nicht angegeben wurde, erzeugt der Compiler ein Array das gerade groß genug ist, um die Werte aus der Initialisierung aufzunehmen. Deshalb ist:

```
int Ary[4][5] = {
            { 10, 11, 12, 13, 14 },
            { 24, 25, 26, 27, 28 },
            { 30, 31, 32, 33, 34 },
            { 44, 45, 46, 47, 48 },
        }
```

das gleiche wie:

```
int Ary[ ][5] = {
            { 10, 11, 12, 13, 14 },
            { 24, 25, 26, 27, 28 },
            { 30, 31, 32, 33, 34 },
            { 44, 45, 46, 47, 48 },
        }
```

Ob man die Größe angibt oder weglässt ist jedem selbst überlassen, jedoch ist es zu empfehlen, sie anzugeben.

Falschhingegen wären:

```
int Ary[5][ ] = {
```

```
            { 10, 11, 12, 13, 14 },
            { 24, 25, 26, 27, 28 },
            { 30, 31, 32, 33, 34 },
            { 44, 45, 46, 47, 48 },
        }
```

oder:

```
int Ary[ ][ ] = {
            { 10, 11, 12, 13, 14 },
            { 24, 25, 26, 27, 28 },
            { 30, 31, 32, 33, 34 },
            { 44, 45, 46, 47, 48 },
        }
```

genau wie:

```
int Ary[ ][4][ ] = {
            { 10, 11, 12, 13, 14 },
            { 24, 25, 26, 27, 28 },
            { 30, 31, 32, 33, 34 },
            { 44, 45, 46, 47, 48 },
        }
```

und:

```
int Ary[ ][ ][5] = {
            { 10, 11, 12, 13, 14 },
            { 24, 25, 26, 27, 28 },
            { 30, 31, 32, 33, 34 },
            { 44, 45, 46, 47, 48 },
        }
```

10.8 Mehrdimensionales Array teilweise initialisiert

```
int Ary[4][5] = {
            { 10, 11, 12, 13, 14 },
            { 24, 25, 26, 27, 28 },
            { 30, 31, 32 },
        }
```

```
Index      | Inhalt
--------------------
Ary[0][0] = 10
Ary[0][1] = 11
Ary[0][2] = 12
Ary[0][3] = 13
Ary[0][4] = 14

Ary[1][0] = 24
Ary[1][1] = 25
Ary[1][2] = 26
Ary[1][3] = 27
Ary[1][4] = 28

Ary[2][0] = 30
Ary[2][1] = 31
```

```
Ary[2][2] = 32
Ary[2][3] = 0
Ary[2][4] = 0

Ary[3][0] = 0
Ary[3][1] = 0
Ary[3][2] = 0
Ary[3][3] = 0
Ary[3][4] = 0
```

Das teilweise Initalisieren eines mehrdimensionalen Arrays folgt genau dem selben Muster wie auch schon beim teilweise initalisierten eindimensionalen Array. Hier werden auch nur die ersten 13 Felder mit dem Index [0|0] bis [2|2] gefüllt. Die restlichen werden vom Compiler mit 0 gefüllt.

Es ist wichtig nicht zu vergessen, dass die Werte aus der Liste den Array-Elementen ab dem Index Nummer Null übergeben werden und nicht erst ab dem Index Nummer Eins! Außerdem kann man auch keine Felder überspringen, um den ersten Wert aus der Liste beispielsweise erst dem fünften oder siebten Array-Element zu übergeben!

Fehlende Größenangabe bei teilweise initialisierten Mehrdimensionalen Arrays

Die Verwendung von teilweise initialisierte mehrdimensionale Arrays mit fehlender Größenangabe macht genauso wenig Sinn wie auch bei teilweise initialiserten Eindimensionalen Array. Denn eine fehlende Größenangabe führt in solch einem Fall dazu, dass die Größe des Arrays womöglich nicht ausreichend ist, weil nur genug Array-Elemente vom Compiler erstellt wurden um die Werte aus der Liste auf zu nehmen. Deshalb sollte man bei solchen niemals vergessen die Größe mit anzugeben.

10.9 Arrays und deren Speicherplatz

Die Elementgröße eines Arrays hängt zum einen vom verwendeten Betriebssystem und zum anderen vom angegebenen Datentyp ab, mit dem das Array deklariert wurde.

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

signed char siAry1[200];
signed short siAry2[200];
signed int siAry3[200];
signed long int siAry4[200];
signed long long int siAry5[200];

unsigned char unAry1[200];
unsigned short unAry2[200];
unsigned int unAry3[200];
unsigned long int unAry4[200];
unsigned long long int unAry5[200];

float Ary6[200];
double Ary7[200];
long double Ary8[200];


int main( void )
{
    printf( "Datentyp des Elements | Byte (Elementgröße) \n" );
```

```
   printf( "Signed: \n" );
   printf( "signed char        =    %d Byte \n", sizeof(signed char) );
   printf( "signed short        =    %d Byte \n", sizeof(signed short) );
   printf( "signed int        =    %d Byte \n", sizeof(signed int) );
   printf( "signed long int        =    %d Byte \n", sizeof(signed long int) );
   printf( "signed long long int    =    %d Byte \n\n", sizeof(signed long long
int) );
   //
   printf( "Unsigned: \n" );
   printf( "unsigned char        =    %d Byte \n", sizeof(unsigned char) );
   printf( "unsigned short        =    %d Byte \n", sizeof(unsigned short) );
   printf( "unsigned int        =    %d Byte \n", sizeof(unsigned int) );
   printf( "unsigned long int    =    %d Byte \n", sizeof(unsigned long int) );
   printf( "unsigned long long int    =    %d Byte \n\n", sizeof(unsigned long
long int) );
   //
   printf( "Signed ohne prefix \n" );
   printf( "float            =    %d Byte \n", sizeof(float) );
   printf( "double            =    %d Byte \n", sizeof(double) );
   printf( "long double        =    %d Byte \n\n\n", sizeof(long double) );


   printf( "Groeße, mit verschiedenen Datentyp, eines arrays mit 200Feldern \n"
);
   printf( "Signed: \n" );
   printf( "Groeße von siAry als signed char = %d Byte \n", sizeof(siAry1) );
   printf( "Groeße von siAry als signed short = %d Byte \n", sizeof(siAry2) );
   printf( "Groeße von siAry als signed int = %d Byte \n", sizeof(siAry3) );
   printf( "Groeße von siAry als signed long int = %d Byte \n", sizeof(siAry4)
);
   printf( "Groeße von siAry als signed long long int = %d Byte \n\n",
sizeof(siAry5) );
   //
   printf( "Unsigned: \n" );
   printf( "Groeße von unAry als unsigned char = %d Byte \n", sizeof(unAry1) );
   printf( "Groeße von unAry als unsigned short = %d Byte \n", sizeof(unAry2)
);
   printf( "Groeße von unAry als unsigned int = %d Byte \n", sizeof(unAry3) );
   printf( "Groeße von unAry als unsigned long int = %d Byte \n",
sizeof(unAry4) );
   printf( "Groeße von unAry als unsigned long long int = %d Byte \n\n",
sizeof(unAry5) );
   //
   printf( "Signed ohne prefix \n" );
   printf( "Groeße von Ary als float = %d Byte \n", sizeof(Ary6) );
   printf( "Groeße von Ary als double = %d Byte \n", sizeof(Ary7) );
   printf( "Groeße von Ary als long double = %d Byte \n\n", sizeof(Ary8) );

   return 0;
}
```

Die Speicherplatzgröße eines gesamten Arrays hängt vom verwendeten Datentyp bei der deklaration und von der Anzahl der Elemente die es beinhaltet ab.

Die maximale Größe eines Array wird nur durch den verfügbaren Speicher limitiert.

```
Den Array-Speicherpaltz ermitteln:

Array Größe           =    [ (Anzahl der Elemente) x (Datentyp) ]
-------------------------------------------------------------------------------
char Ary[500]            |    [ 500(Elemente) x 1(Typ.Größe) ] =  500 Byte
short Ary[500]           |    [ 500(Elemente) x 2(Typ.Größe) ] = 1000 Byte
int Ary[500]             |    [ 500(Elemente) x 4(Typ.Größe) ] = 2000 Byte
long int Ary[500]        |    [ 500(Elemente) x 4(Typ.Größe) ] = 2000 Byte
long long int Ary[500]   |    [ 500(Elemente) x 8(Typ.Größe) ] = 4000 Byte
```

```
float Ary[500]           |    [ 500(Elemente) x 4(Typ.Größe) ] = 2000 Byte
double Ary[500]          |    [ 500(Elemente) x 8(Typ.Größe) ] = 4000 Byte
long double Ary[500]     |    [ 500(Elemente) x 12(Typ.Größe)] = 6000 Byte
-------------------------------------------------------------------------
```

Anmerkung: Bei einem 64bit System unterscheiden sich die Werte.

10.10 Übergabe eines Arrays an eine Funktion

Bei der Übergabe von Arrays an Funktionen wird nicht wie bei Variablen eine Kopie übergeben, sondern immer ein Zeiger auf das Array. Der Grund hierfür besteht im Zeitaufwand: Würde die Funktion mit einer Kopie arbeiten, so müsste jedes einzelne Element kopiert werden.

Das folgende Beispielprogramm zeigt die Übergabe eines Arrays an eine Funktion:

```
#include <stdio.h>

void function( int feld[ ] )
{
    feld[1] = 10;
    feld[3] = 444555666;
    feld[8] = 25;
}

int main( void )
{
    int feld[9] = { 1, 2, 3, 4, 5, 6 };
    printf( "Der Inhalt des fuenften array Feldes ist: %d \n", feld[4] );
    printf( "Der Inhalt des sechsten array Feldes ist: %d \n\n", feld[5] );

    function( feld );
    printf( "Der Inhalt des ersten array Feldes ist: %d \n", feld[0]);
    printf( "Der Inhalt des zweiten array Feldes ist: %d \n", feld[1] );
    printf( "Der Inhalt des dritten array Feldes ist: %d \n", feld[2]);
    printf( "Der Inhalt des vierte array Feldes ist: %d \n", feld[3]);
    printf( "Der Inhalt des fuenften array Feldes ist: %d \n", feld[4] );
    printf( "Der Inhalt des neunten array Feldes ist: %d \n\n", feld[8] );

    printf( "Inhalt des nicht existierenden 10 array Feldes ist: %d \n", feld[9]
 );
    printf( "Inhalt des nicht existierenden 11 array Feldes ist: %d \n",
 feld[10] );
    printf( "Inhalt des nicht existierenden 12 array Feldes ist: %d \n",
 feld[11] );
    printf( "Inhalt des nicht existierenden 26 array Feldes ist: %d \n",
 feld[25] );
    printf( "Inhalt des nicht existierenden 27 array Feldes ist: %d \n",
 feld[26] );
    return 0;
}
```

Nach dem Ausführen erhalten Sie als Ausgabe:

```
Der Inhalt des fuenften array Feldes ist: 5
Der Inhalt des sechsten array Feldes ist: 6

Der Inhalt des ersten array Feldes ist: 1
Der Inhalt des zweiten array Feldes ist: 10
Der Inhalt des dritten array Feldes ist: 3
```

```
Der Inhalt des vierte array Feldes ist: 444555666
Der Inhalt des fuenften array Feldes ist: 5
Der Inhalt des neunten array Feldes ist: 25

Inhalt des nicht existierenden 10 array Feldes ist: 134514000
Inhalt des nicht existierenden 11 array Feldes ist: 134513424
Inhalt des nicht existierenden 12 array Feldes ist: -1080189048
Inhalt des nicht existierenden 26 array Feldes ist: -2132909840
Inhalt des nicht existierenden 27 array Feldes ist: 2011993312
```

Da das Array nur neun Felder besitzt geben alle Felder ab zehn nur noch Fehlerwerte zurück.

Mit dem Funktionsaufruf

```
function( feld );
```

wird ein Zeiger auf das erste Element des Arrays an das Unterprogramm übergeben. Ausgehend von der Adresse des ersten Elements können die Adressen der nächsten Elemente berechnet werden und somit auf die Werte der Elemente zugegriffen werden. Die Adressberechnung erfolgt automatisch im Hintergrund zum Beispiel durch den Zugriff mittels feld[1].

Hier zwei gleichbedeutende Schreibweisen:

```
int feld[]
int *feld
```

Beide Schreibweisen stellen die Definition eines Zeigers dar.

Die alternative Darstellungsform ließe sich also wie folgt realisieren:

```
#include <stdio.h>

void function( int *feld )
{
    feld[1] = 10;
    feld[3] = 444555666;
    feld[8] = 25;
}

int main( void )
{
    int feld[9] = { 1, 2, 3, 4, 5, 6 };
    printf( "Der Inhalt des fuenften array Feldes ist: %d \n", feld[4] );
    printf( "Der Inhalt des sechsten array Feldes ist: %d \n\n", feld[5] );

    function( feld );
    printf( "Der Inhalt des ersten array Feldes ist: %d \n", feld[0]);
    printf( "Der Inhalt des zweiten array Feldes ist: %d \n", feld[1] );
    printf( "Der Inhalt des dritten array Feldes ist: %d \n", feld[2]);
    printf( "Der Inhalt des vierte array Feldes ist: %d \n", feld[3]);
    printf( "Der Inhalt des fuenften array Feldes ist: %d \n", feld[4] );
    printf( "Der Inhalt des neunten array Feldes ist: %d \n\n", feld[8] );

    printf( "Inhalt des nicht existierenden 10 array Feldes ist: %d \n", feld[9]
 );
    printf( "Inhalt des nicht existierenden 11 array Feldes ist: %d \n",
 feld[10] );
    printf( "Inhalt des nicht existierenden 12 array Feldes ist: %d \n",
 feld[11] );
    printf( "Inhalt des nicht existierenden 26 array Feldes ist: %d \n",
 feld[25] );
    printf( "Inhalt des nicht existierenden 27 array Feldes ist: %d \n",
 feld[26] );
```

```
    return 0;
}
```

Mehrdimensionale Arrays übergeben Sie entsprechend der Dimensionszahl wie eindimensionale. [] und * lassen sich auch hier in geradezu abstrusen Möglichkeiten vermischen, doch dabei entsteht unleserlicher Programmcode. Hier eine korrekte Möglichkeit, ein zweidimensionales Feld an eine Funktion zu übergeben:

```
#include <stdio.h>

void function( int feld[2][5] )
{
    feld[1][2] = 55;
}

int main( void )
{
    int feld[2][5] = {
                        { 10, 11, 12, 13, 14 },
                        { 20, 21, 22, 23, 24 }
                     };

    printf( "%d \n", feld[1][2] );

    function( feld );
    printf( "%d \n", feld[1][2] );

    return 0;
}
```

10.11 Zeigerarithmetik

Auf Zeiger können auch arithmetische Operatoren wie der Additions- und der Subtraktionsoperator sowie die Vergleichsoperatoren angewendet werden. Dagegen ist die Verwendung von anderen Operatoren wie beispielsweise dem Multiplikations- oder Divisionsoperator nicht erlaubt.

Die Operatoren können verwendet werden, um innerhalb eines Arrays auf verschiedene Elemente zuzugreifen, oder die Position innerhalb des Arrays zu vergleichen. Hier ein kurzes Beispiel um es zu verdeutlichen:

```
#include <stdio.h>

int main( void )
{
    int *ptr;
    int a[5] = { 1, 2, 3, 5, 7 };


    ptr = &a[0];
    printf( "a) Die Variable enthält den Wert: %d \n", *ptr );
    //
    ptr += 2;
    printf( "b) Nach der addition enthält die Variable den Wert: %d \n", *ptr );
    //
    ptr -= 1;
    printf( "c) Nach der subtraktion enthält die Variable den Wert: %d \n", *ptr
 );
    //
```

```
    ptr += 3;
    printf( "d) Nach der addition enthält die Variable den Wert: %d \n", *ptr );
    //
    ptr -= 1;
    printf( "e) Nach der subtraktion enthält die Variable den Wert: %d \n", *ptr
 );
    return 0;
}
```

Wir deklarieren einen Zeiger sowie ein Array und weisen dem Zeiger die Adresse des ersten Elementes zu (Abb. 2). Da der Zeiger der auf das erste Element im Array gerichtet ist äquivalent zum Namen des Array ist, kann man diesen auch kürzen. Deshalb ist:

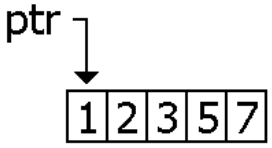


Abb. 5 Abb. 2

```
ptr = &a[0];
```

das gleiche wie:

```
ptr = a;
```

Auf den Zeiger ptr kann nun beispielsweise der Additionsoperator angewendet werden. Mit dem Ausdruck

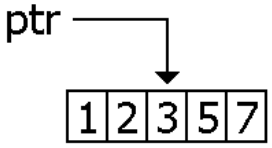


Abb. 6 Abb. 3

```
ptr += 2
```

wird allerdings nicht etwa `a[0]` erhöht, sondern `ptr` zeigt nun auf `a[2]` (Abb. 3).

Wenn `ptr` auf ein Element des Arrays zeigt, dann zeigt `ptr += 1` auf das nächste Element, `ptr += 2` auf das übernächste Element usw. Wendet man auf einen Zeiger den Dereferenzierungsoperator(*) an, so erhält man den Inhalt des Elements, auf das der Zeiger gerade zeigt. Wenn beispielsweise `ptr` auf `a[2]` zeigt, so entspricht `*ptr` dem Wert des dritten Elements des Arrays.

Auch Inkrement- und Dekrementoperator können auf Zeiger auf Vektoren angewendet werden. Wenn `ptr` auf `a[2]` zeigt, so erhält man über `ptr++` die Adresse des Nachfolgeelements `a[3]`. Hier ein weiteres Beispiel um es zu veranschaulichen:

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

```
int main( void )
{
    int *ptr;
    int a[5] = { 1, 2, 3, 5, 7 };

    ptr = &a[0];
    printf( "a) Die Variable enthält den Wert: %d \n", *ptr );
        //
        ptr += 2;
    printf( "b) Nach der addition enthält die Variable den Wert: %d \n", *ptr );
        //
        ptr -= 1;
    printf( "c) Nach der subtraktion enthält die Variable den Wert: %d \n", *ptr
 );
        //
        ptr += 3;
    printf( "d) Nach der addition enthält die Variable den Wert: %d \n", *ptr );
        //
        ptr -= 1;
    printf( "e) Nach der subtraktion enthält die Variable den Wert: %d \n", *ptr
 );


    ptr--;
    printf( "a) Nach der subtraktion enthält die Variable den Wert: %d \n", *ptr
 );
    //
    --ptr;
    printf( "b) Nach der subtraktion enthält die Variable den Wert: %d \n", *ptr
 );
    //
    ptr++;
    printf( "c) Nach der addition enthält die Variable den Wert: %d \n", *ptr );
    //
    ++ptr;
    printf( "d) Nach der addition enthält die Variable den Wert: %d \n", *ptr );
    return 0;
}
```

Um die neue Adresse berechnen zu können, muss der Compiler die Größe des Zeigertyps kennen. Deshalb ist es nicht möglich, die Zeigerarithmetik auf den Typ `void*`anzuwenden.

Grundsätzlich ist zu beachten, dass der `[]`-Operator in C sich aus den Zeigeroperationen heraus definiert.

Daraus ergeben sich recht kuriose Möglichkeiten in C: So ist `a[b]`als `*(a+b)`definiert, was wiederum gleichbedeutend ist mit `*(b+a)`und man somit nach Definition wieder als `b[a]`schreiben kann. So kommt es, dass ein 4[a] das gleiche Ergebnis liefert, wie a[4], nämlich das 5. Element vom Array `a`. Das Beispiel sollte man allerdings nur zur Verdeutlichung der Bedeutung des `[]`-Operators verwenden und nicht wirklich anwenden.

10.12 Zeigerarithmetik auf Char-Arrays

Die Zeigerarithmetik bietet natürlich auch eine Möglichkeit, char-Arrays zu verarbeiten. Ein Beispiel aus der Kryptografie[3] verdeutlicht das Prinzip:

3 http://de.wikipedia.org/wiki/Transposition%20%28Kryptographie%29

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

char satz[1024];
char *p_satz;
int satzlaenge;
char neuersatz[1024];
char *p_neuersatz;

int main( void )
{
    fgets( satz, 1024, stdin );
    p_neuersatz = neuersatz;

    for( p_satz = satz; p_satz < satz + ( strlen(satz)-1 ); p_satz += 2 )
    {
        *p_neuersatz = *p_satz;
        ++p_neuersatz;
    }

    for( p_satz = satz+1; p_satz < satz + ( strlen(satz)-1 ); p_satz += 2 )
    {
        *p_neuersatz = *p_satz;
        ++p_neuersatz;
    }

    printf( "Original Satz: %s \n", satz );
    printf( "Verschluesselter Satz: %s \n", neuersatz );
    printf( "Der String ist %d Zeichen lang \n", strlen(satz)-1 );
    return 0;
}
```

Sehen wir uns das Beispiel mal etwas genauer an. Als erstes wird der zusätzlich benötigte Header 'string.h' eingebunden um die Funktion 'strlen' nutzen zu können. Da 'strlen' eine Funktion zum Messen der Länge einer Zeichenkette und kein Operator wie sizeof( ) ist, der die Größe einer Variablen liefert, sollte man beide nicht verwechseln. In Zeile 12 wird dann mit der Funktion 'fgets' über die Standardeingabe (stdin) ein Zeichenkette von maximal 1024 Zeichen entgegen genommen und diese im Array 'satz' abgelegt. Anschließend wird noch das Array 'neuersatz' in der Zeigervariablen 'p_neuersatz' abgelegt, um im weiterem Verlauf mit Zeigern auf die Array-Felder zugreifen zu können.

Der Pointer auf 'p_satz' wird sich solange um zwei verschieben, wie der Wert von 'p_satz' kleiner ist als die Länge der Zeichenkette von 'satz'. Zugleich wird pro Schleifendurchlauf jeweils ein einzelnes Zeichen als Variable an die Zeigervariable 'p_neuersatz', die ein eigenständiges Array mit 1024 Elementen beinhaltet, übergeben und anschließend der Pointer dieser Zeigervaraible erhöht, um auf das nächste Array-Element zeigen zu können. Ist der Pointer erhöht kann dann das nächste Zeichen entgegen genommen, abgelegt und anschließend der Pointer wieder um eins angehoben/verschoben werden.

Hinweis: *Würde man die Durchläufe der Schleife nicht auf die Länge der eingegebenen Zeichenkette beschränken würde diese bis zu einem Pointer Überlauf durchlaufen, was dann wiederum zu unerwarteten Fehlern führen könnte.*

10.13 Strings

C besitzt im Gegensatz zu vielen anderen Sprachen keinen Datentyp für Strings (Zeichenketten). Stattdessen werden für Zeichenketten Arrays[4] verwendet. Das Ende des Strings ist durch das sogenannte String-Terminierungszeichen *\0*gekennzeichnet. Beispielsweise wird über

`const char text[5]="Wort";`oder `const char text[]="Wort";`

jeweils ein String definiert. Ausführlich geschrieben entsprechen die Definitionen

```
char text[5];
text[0]='W';
text[1]='o';
text[2]='r';
text[3]='t';
text[4]='\0';
```

Zu beachten ist dabei, dass einzelne Zeichen mit Hochkommata (') eingeschlossen werden müssen. Strings dagegen werden immer mit Anführungszeichen (") markiert. Im Gegensatz zu 'W' in Hochkommata entspricht "W" dem Zeichen 'W' und zusätzlich dem Terminierungszeichen '\0'.

en:C Programming/Arrays[5] it:Linguaggio C/Vettori e puntatori/Vettori[6] pl:C/Tablice[7]

4 Kapitel 10.1 auf Seite 85
5 http://en.wikibooks.org/wiki/C%20Programming%2FArrays
6 http://it.wikibooks.org/wiki/Linguaggio%20C%2FVettori%20e%20puntatori%2FVettori
7 http://pl.wikibooks.org/wiki/C%2FTablice

11 Strings und Zeichenkettenfunktionen

11.1 Zeichenkettenfunktionen

Für die Bearbeitung von Strings stellt C eine Reihe von Bibliotheksfunktionen zu Verfügung:

11.1.1 strcpy

```
char* strcpy(char* Ziel, const char* Quelle)
```

Kopiert einen String in einen anderen (Quelle nach Ziel) und liefert Zeiger auf Ziel als Funktionswert. Bitte beachten Sie, dass eine Anweisung `text2 = text1`für ein Array nicht möglich ist. Für eine Kopie eines Strings in einen anderen ist immer die Anweisung `strcpy`nötig, da eine Zeichenkette immer zeichenweise kopiert werden muss.

Beispiel:

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

int main(void)
{
  char text[20];

  strcpy(text, "Hallo!");
  printf("%s\n", text);
  strcpy(text, "Ja Du!");
  printf("%s\n", text);
  return 0;
}
```

Nach dem Übersetzen und Ausführen erhält man die folgende Ausgabe:

```
  Hallo!
  Ja Du!
```

11.1.2 strncpy

```
char* strncpy(char* Ziel, const char* Quelle, size_t num)
```

Kopiert `num`-Zeichen von `Quelle`zu `Ziel`. Wenn das Ende des `Ziel`C-String (welches ein null-Character (`'\0'`) signalisiert) gefunden wird, bevor `num`-Zeichen kopiert sind, wird Ziel mit `'\0'`-Zeichen aufgefüllt bis die komplette Anzahl von `num`-Zeichen in `Quelle`geschrieben ist.

Wichtig: `strncpy()`fügt selbst keinen null-Character (`'\0'`) an das Ende von `Ziel`. Soll heißen: `Ziel`wird nur null-Terminiert wenn die Länge des C-Strings `Quelle`kleiner ist als `num`.

Beispiel:

```
/* strncpy Beispiel*/
#include <stdio.h>
#include <string.h>

int main ()
{
    char strA[] = "Hallo!";
    char strB[6];

    strncpy(strB, strA, 5);

    /* Nimm die Anzahl von Bytes in strB (6), ziehe 1 ab (= 5) um auf den
 letzten index zu kommen,
       dann füge dort ein null-Terminierer ein. */
    strB[sizeof(strB)-1] = '\0';

    puts(strB);

    return 0;
}
```

Vorsicht: Benutzen Sie `sizeof()`in diesem Zusammenhang nur bei Character Arrays. `sizeof()`gibt die Anzahl der reservierten Bytes zurück. In diesem Fall: `6`(Größe von strB) `* 1 Byte`(Character) `= 6`.

Nach dem Übersetzen und Ausführen erhält man die folgende Ausgabe:

```
Hallo
```

11.1.3 strcat

```
char* strcat(char* s1, const char* s2)
```

Verbindet zwei Zeichenketten miteinander. Natürlich wird das Stringende-Zeichen \0 von s1 überschrieben. Voraussetzung ist, dass s2 in s1 Platz hat.

Beispiel:

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

int main(void)
{
  char text[20];
  strcpy(text, "Hallo!");
  printf("%s\n", text);
  strcat(text, "Ja du!");
  printf("%s\n", text);
  return 0;
}
```

Nach dem Übersetzen und Ausführen erhält man die folgende Ausgabe:

```
Hallo!
Hallo!Ja du!
```

Wie Sie sehen wird der String in Zeile 9 diesmal nicht überschrieben, sondern am Ende angehängt.

11.1.4 strncat

Eine sichere Variante ist strncat:

```
char* strncat(char* s1, const char* s2,size_t n)
```

Es werden nur n Elemente angehängt. Damit wird sichergestellt, dass nicht in einen undefinierten Speicherbereich geschrieben wird.

In jedem Fall wird jedoch ein '\0' - Zeichen an das Ende der Resultat-Zeichenfolge angehängt.

Soweit nicht anders angegeben, ist das Ergebnis für überlappende Bereiche **nicht**definiert.

Anmerkung:n ist so zu wählen, das die Länge der Resultat-Zeichenkette s1 nicht überschritten wird.

11.1.5 strtok

```
char *strtok( char *s1, const char *s2 )
```

Die Funktion strtok zerlegt einen String(*s1*) in einzelne Teilstrings anhand von sogenannten Token. Der String wird dabei durch ein Trennzeichen(*s2*) getrennt.

"s2" kann mehrere Trennzeichen enthalten, z.B. s2=" ,\n." (d.h. Trennung bei Space, Komma, New-Line, Punkt).

Der Code zu diesem Beispiel würde folgendermaßen aussehen:

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

int main(void) {
    char text[] = "Das ist ein Beispiel!";
    char trennzeichen[] = " ";
    char *wort;
    int i=1;
    wort = strtok(text, trennzeichen);

    while(wort != NULL) {
        printf("Token %d: %s\n", i++, wort);
        wort = strtok(NULL, trennzeichen);

        //Jeder Aufruf gibt das Token zurück. Das Trennzeichen wird mit '\0'
 überschrieben.
        //Die Schleife läuft durch bis strtok() den NULL-Zeiger zurückliefert.
```

```
    }
    return 0;
}
```

```
Token1: Das
Token2: ist
Token3: ein
Token4: Beispiel!
```

11.1.6 strcspn

```
int strcspn (const *s1, const *s2)
```

strcspn() ist eine Funktion der Standardbibliothek string.h die, je nach Compiler, in jedes C-Programm implementiert werden kann. strcspn() dient dazu, die Stelle eines Zeichens zu ermitteln, an der es zuerst in einem String vorkommt.

Sobald ein Zeichen aus s2 in s1 gefunden wird, wird der Wert der Position an der es gefunden wurde, zurückgegeben.

Achtung:es wird der Index im Char-Array ermittelt.

Beispiel:

```
/*Beispiel zu strcspn()*/

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

int main(void){
    char s1[] = "Das ist ein Text";
    char s2[] = "tbc";           // gesucht werden soll das Zeichen t, b oder c
    int position = 0;

    position = strcspn(s1,s2);
    printf("Das erste Zeichen von %s tritt an der Stelle %i auf.\n", s2,
 position+1);
    // Da der Positionszähler der Zeichenkette bei 0 (wie bei Arrays) beginnt,
    // muss zur gefundenen Position 1 zuaddiert werden um eine korrekte
    // Darstellung zu erhalten.

    return 0;
}
```

```
Das erste Zeichen von tbc tritt an der Stelle 7 auf.
```

11.1.7 strpbrk

```
char *strpbrk( const char *string1, const char *string2);
```

Gibt einen Zeiger auf das erste Zeichen in string1 zurück, das auch in string2 enthalten ist. Es wird also nicht nach einer Zeichenkette gesucht, sondern nach einem einzelnen Zeichen aus einer Zeichen*menge*. War die Suche erfolglos, wird NULL zurückgegeben.

Ein Beispiel:

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

int main()
{
   char str1[]="Schwein gehabt!";
   char str2[]="aeiou";
   printf("%s\n", strpbrk(str1, str2));
   return 0;
}
```

Ausgabe:

```
ein gehabt!
```

11.1.8 strrchr

```
char *strrchr(const char *s, int ch)
```

strrchr() ist eine Stringfunktion, die das letzte Auftreten eines Zeichens in einer Zeichenkette sucht. Als Ergebnis wird der Zeiger auf dieses Zeichen geliefert bzw. der NULL-Zeiger, falls dieses Zeichen nicht gefunden wurde.

Im folgenden Beispiel wird eine Zeichenkette mit fgets eingelesen. fgets hängt am Ende ein New-Line-Zeichen an (\n). Wir suchen mit einem Zeiger nach diesem Zeichen und ersetzen es durch ein '\0'-Zeichen.

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

int main()
{
   char string[20];
   char *ptr;

   printf("Eingabe machen:\n");
   fgets(string, 20 , stdin);
   /* man setzt den zeiger auf das New-Line-Zeichen */
   ptr = strrchr(string, '\n');
   /* \n-Zeichen mit \0 überschreiben */
   *ptr = '\0';
   printf("%s\n",string);
   return 0;
}
```

Ein weiteres Beispiel:

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

int main()
{
   char string[]="Dies ist wichtig. Dies ist nicht wichtig";
   char *ptr;
   // suche Trennzeichen "." vom Ende der Zeichenkette aus.
   ptr = strrchr (string, '.');
   // wenn Trennzeichen im Text nicht vorhanden,
   // dann ist der Pointer NULL. D.h. NULL muss abgefangen werden.
```

```
   if (ptr != NULL) {
       *ptr = '\0';
   }
   printf ("%s", string);
}
```

Ausgegeben wird:

```
Dies ist wichtig
```

Der Pointer **ptr**zeigt nach **strrchr**genau auf die Speicherstelle des Strings in der das erste Trennzeichen von hinten steht. Wenn man nun an diese Speicherstelle das Zeichenketteendezeichen **0**schreibt, dann ist der String für alle Stringfunktionen an dieser Stelle beendet. **printf**gibt den String **string**nur bis zum Zeichenketteendezeichen aus.

11.1.9 strcmp

```
int strcmp(char* s1, char* s2);
```

Diese Stringfunktion ist für den Vergleich von zwei Strings zu verwenden. Die Strings werden Zeichen für Zeichen durchgegangen und ihre ASCII-Codes verglichen. Wenn die beiden Zeichenketten identisch sind, gibt die Funktion den Wert 0 zurück. Sind die Strings unterschiedlich, gibt die Funktion entweder einen Rückgabewert größer oder kleiner als 0 zurück: Ein Rückgabewert >0 (<0) bedeutet, der erste ungleiche Buchstabe in s1 hat einen größeren (kleineren) ASCII-Code als der in s2.

Beispiel:

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

int main()
{
  const char string1[] = "Hello";
  const char string2[] = "World";
  const char string3[] = "Hello";

  if (strcmp(string1,string2) == 0)
  {
    printf("Die beiden Zeichenketten %s und %s sind
 identisch.\n",string1,string2);
  }
  else
  {
    printf("Die beiden Zeichenketten %s und %s sind
 unterschiedlich.\n",string1,string2);
  }

  if (strcmp(string1,string3) == 0)
  {
    printf("Die beiden Zeichenketten %s und %s sind
 identisch.\n",string1,string3);
  }
  else
  {
    printf("Die beiden Zeichenketten %s und %s sind
 unterschiedlich.\n",string1,string3);
  }
```

```
  return 0;
}
```

Nach dem Ausführen erhält man folgende Ausgabe:

```
  Die beiden Zeichenketten Hello und World sind unterschiedlich.
  Die beiden Zeichenketten Hello und Hello sind identisch.
```

11.1.10 strncmp

```
int strncmp(const char *x, const char *y, size_t n);
```

Diese Funktion arbeitet ähnlich wie die Funktion strcmp(), mit einem Unterschied, dass n Zeichen miteinander verglichen werden. Es werden die ersten n Zeichen von x und die ersten n Zeichen von y miteinander verglichen. Der Rückgabewert ist dabei derselbe wie schon bei strcmp().

Ein Beispiel:

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

int main()
{
   const char x[] = "aaaa";
   const char y[] = "aabb";
   int i;

   for(i = strlen(x); i > 0; --i)
      {
         if(strncmp( x, y, i) != 0)
            printf("Die ersten %d Zeichen der beiden Strings "\
                   "sind nicht gleich\n", i);
         else
            {
               printf("Die ersten %d Zeichen der beiden Strings "\
                      "sind gleich\n", i);
               break;
            }
      }
   return 0;
}
```

```
Die ersten 4 Zeichen der beiden Strings sind nicht gleich
Die ersten 3 Zeichen der beiden Strings sind nicht gleich
Die ersten 2 Zeichen der beiden Strings sind gleich
```

11.1.11 strspn

Die Funktion strspn() gibt die Position des ersten Vorkommens eines Zeichens an, das nicht vorkommt. Die Syntax lautet:

```
int strspn(const char *s1, const char *s2);
```

Folgendes Beispiel gibt Ihnen die Position des Zeichens zurück, welches keine Ziffer ist:

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

int main()
{
   const char string[] = "7501234-123";
   int pos = strspn(string, "0123456789");
   printf("Die Position, an der keine Ziffer steht:");
   printf(" %d\n",++pos);
   // pos wird um 1 erhöht, da die Positionierung mit 0 anstatt 1 beginnt
   return 0;
}
```

```
Die Position, an der keine Ziffer steht: 8
```

11.1.12 strchr

```
char* strchr(char * string, int zeichen)
```

Die Funktion strchr (string char) sucht das erste Vorkommen eines Zeichens in einem String. Sie liefert entweder die Adresse des Zeichens zurück oder NULL, falls das Zeichen nicht im String enthalten ist.

Beispiel:

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

int main()
{
  char string[] = "Ein Teststring mit Worten";
  printf("%s\n",strchr(string, (int)'W'));
  printf("%s\n",strchr(string, (int)'T'));
  return 0;
}
```

Hier die Ausgabe des Programms:

```
Worten
Teststring mit Worten
```

Noch ein Beispiel:

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

int main()
{
  char string[]="Dies ist wichtig. Dies nicht.";
  char *stelle;

  stelle=strchr(string, (int)'.');
  *(stelle+1)='\0';/*Durch *(stelle+1) wird nicht der Punkt,
                     sondern das Leerzeichen (das Zeichen danach)
                     durch das Determinierungszeichen ersetzt*/
  printf("%s", string);
```

```
  return 0;
}
```

Hier die Ausgabe des Programms:

```
Dies ist wichtig.
```

Achtung: Achten Sie auf die richtige Rechtschreibung, da aus strchr leicht strrchr werden kann. Ansonsten wird statt des ersten, das letzte Zeichen im String ausgelesen.

11.1.13 strlen

Zum Ermitteln der Länge eines String, kann man die Funktion strlen() verwenden. Die Syntax lautet wie folgt:

```
size_t strlen(const char *string1)
```

Mit dieser Syntax wird die Länge des adressierten Strings string1 ohne das Determinierungszeichen zurückgegeben.

Nun ein Beispiel zu strlen():

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

int main()
{
  char string1[] = "Das ist ein Test";
  size_t length;

  length = strlen(string1);
  printf("Der String \"%s\" hat %d Zeichen\n", string1, length);

  return 0;
}
```

```
Der String "Das ist ein Test" hat 16 Zeichen
```

11.1.14 strstr

```
char *strstr(const char *s1, const char *s2);
```

Sucht das erste Vorkommen der Zeichenfolge s2 (ausschließlich des abschließenden '\0' - Zeichens) in der Zeichenfolge s1. Als Ergebnis wird ein Zeiger auf die gefundene Zeichenfolge (innerhalb s1) geliefert bzw. der NULL-Zeiger, falls die Suche erfolglos war. Ist die Länge der Zeichenfolge s2 0, so wird der Zeiger auf s1 geliefert.

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

int main ()
{
```

```
  char str[] = "Dies ist ein simpler string";
  char *ptr;
  // setzt den Pointer ptr an die Textstelle "simpler"
  ptr = strstr (str, "simpler");
  // ersetzt den Text an der Stelle des Pointers mit "Beispiel"
  strncpy (ptr, "Beispiel", 8);
  puts (str);
  return 0;
}
```

```
Dies ist ein Beispielstring
```

11.2 Gefahren

Bei der Verarbeitung von *Strings*muss man sehr vorsichtig sein, um nicht über das Ende eines Speicherbereiches hinauszuschreiben oder zu -lesen. Generell sind Funktionen wie `strcpy()`und `sprintf()`zu vermeiden und stattdessen `strncpy()`und `snprintf()`zu verwenden, weil dort die Größe des Speicherbereiches angegeben werden kann.

```
#include <string.h>   // fuer Zeichenketten-Manipulation
#include <stdio.h>    // fuer printf()

int main(void)
{
  char text[20];

  strcpy(text, "Dies ist kein feiner Programmtest"); //Absturzgefahr, da
 Zeichenkette zu lang

  strncpy(text, "Dies ist ein feiner Programmtest", sizeof(text));
  printf("Die Laenge ist %u\n", strlen(text)); //Absturzgefahr, da Zeichenkette
 'text' nicht terminiert

  // also vorsichtshalber mit \0 abschliessen.
  text[sizeof(text)-1] = '\0';
  printf("Die Laenge von '%s' ist %u \n", text, strlen(text));

  return 0;
}
```

Die beiden Zeilen 8 und 11 bringen das Programm möglicherweise zum Absturz:

- **Zeile 8:**`strcpy()`versucht mehr Zeichen zu schreiben, als in der Variable vorhanden sind, was möglicherweise zu einem Speicherzugriffsfehler führt.
- **Zeile 11:**Falls das Programm in Zeile 8 noch nicht abstürzt, geschieht das evtl. jetzt. In Zeile 10 werden genau 20 Zeichen kopiert, was prinzipiell in Ordnung ist. Weil aber der Platz nicht ausreicht, wird die abschließende \0 ausgespart, was bedeutet, dass die Zeichenkette nicht terminiert ist. Die Funktion `strlen()`benötigt aber genau diese '\0', um die Länge zu bestimmen. Tritt dieses Zeichen nicht auf, kann es zu einem Speicherzugriffsfehler kommen.

Entfernt man die beiden Zeilen 8 und 11 ergibt sich folgende Ausgabe:

```
Die Laenge von 'Dies ist ein feiner' ist 19
```

Es ist klar, dass sich hier als Länge 19 ergibt, denn ein Zeichen wird eben für die 0 verbraucht. Man muss also immer daran denken, ein zusätzliches Byte dafür einzurechnen.

11.3 Iterieren durch eine Zeichenkette (Ersetzen eines bestimmten Zeichens durch ein anderes in einem String)

```
#include <string.h>   // fuer Zeichenketten-Manipulation
#include <stdio.h>    // fuer printf()

 /* Diese Funktion ersetzt in einer Zeichenkette ein Zeichen
  * durch ein anderes. Der Rückgabewert ist die Anzahl der
  * Ersetzungen */
unsigned replace_character(char* string, char from, char to)
{
  unsigned result = 0;

  if (!string) return 0;

  while (*string != '\0')
  {
    if (*string == from)
    {
      *string = to;
      result++;
    }
    string++;
  }
  return result;
}

int main(void)
{
  char text[50] = "Dies ist ein feiner Programmtest";
  unsigned result;

  result = replace_character(text, 'e', ' ');
  printf("%u Ersetzungen: %s\n", result, text);

  result = replace_character(text, ' ', '#');
  printf("%u Ersetzungen: %s\n", result, text);

  return 0;
}
```

Die Ausgabe lautet:

```
5 Ersetzungen: Di s ist  in f in r Programmt st
9 Ersetzungen: Di#s#ist##in#f#in#r#Programmt#st
```

Zuerst wird die Funktion replace_character deklariert, mit der ein gewisses Zeichen in einem String durch ein anderes ersetzt wird. Grundsätzlich wird die Funktion solange wiederholt, bis sie das Determinierungszeichen \0 findet. Dabei vergleicht sie das erste Zeichen (dessen Adresse bei der Parameterrückgabe mit dem Namen zurückgegeben wird, da der Name auch immer die Adresse des ersten Zeichens ist) mit dem From-Zeichen. Falls beide Zeichen identisch sind, wird die Stelle im String, auf dem der Handle gerade zeigt, durch das to-

Zeichen ersetzt. Außerdem wird die Variablen result um eins vergrößert und dann am Schluss zurückgegeben.

11.4 Die Bibliothek `ctype.h`

Wie wir bereits im Kapitel Variablen und Konstanten[1] gesehen haben, sagt der Standard nichts über den verwendeten Zeichensatz aus. Nehmen wir beispielsweise an, wir wollen testen, ob in der Variable `c`ein Buchstabe gespeichert ist. Dazu verwenden wir die Anweisung

```
if ('A' <= c && c <= 'Z' || 'a' <= c && c <= 'z')
```

Wenn in c ein Zeichen, das zwischen den Buchstaben A und Z liegt, bzw. den Buchstaben a und z, dann ...

Unglücklicherweise funktioniert das Beispiel zwar auf dem ASCII-Zeichensatz, nicht aber mit dem EBCDIC-Zeichensatz. Der Grund hierfür ist, dass die Buchstaben beim EBCDIC-Zeichensatz nicht hintereinander stehen.

Wer eine plattformunabhängige Lösung sucht, kann deshalb auf Funktionen der Standardbibliothek zurückgreifen. Ihre Prototypen sind alle in der Headerdatei `<ctype.h>`definiert. Für den Test auf Buchstabe können wir die Funktion `int isalpha(int c)`benutzen. Alle Funktionen, die in der Headerdatei `ctype.h`deklariert sind, liefern einen Wert ungleich 0 zurück wenn die entsprechende Bedingung erfüllt ist, andernfalls liefern sie 0 zurück.

Weitere Funktionen von `ctype.h`sind:

- `int isalnum(int c)`testet auf alphanumerisches Zeichen (a-z, A-Z, 0-9)
- `int isalpha(int c)`testet auf Buchstabe (a-z, A-Z)
- `int iscntrl(int c)`testet auf Steuerzeichen (\f, \n, \t ...)
- `int isdigit(int c)`testet auf Dezimalziffer (0-9)
- `int isgraph(int c)`testet auf druckbare Zeichen
- `int islower(int c)`testet auf Kleinbuchstaben (a-z)
- `int isprint(int c)`testet auf druckbare Zeichen ohne Leerzeichen
- `int ispunct(int c)`testet auf druckbare Interpunktionszeichen
- `int isspace(int c)`testet auf Zwischenraumzeichen (Leerzeichen, \f, \n, \t ...)
- `int isupper(int c)`testet auf Großbuchstaben (A-Z)
- `int isxdigit(int c)`testet auf hexadezimale Ziffern (0-9, a-f, A-F)
- `int isblank(int c)`testet auf Leerzeichen

Zusätzlich sind noch zwei Funktionen für die Umwandlung in Groß- bzw. Kleinbuchstaben definiert:

- `int tolower(int c)`wandelt Groß- in Kleinbuchstaben um
- `int toupper(int c)`wandelt Klein- in Großbuchstaben um

en:C Programming/Arrays[2] it:Linguaggio C/Vettori e puntatori/Vettori[3] pl:C/Tablice[4]

1 Kapitel 2.3 auf Seite 15
2 `http://en.wikibooks.org/wiki/C%20Programming%2FArrays`
3 `http://it.wikibooks.org/wiki/Linguaggio%20C%2FVettori%20e%20puntatori%2FVettori`
4 `http://pl.wikibooks.org/wiki/C%2FTablice`

12 Komplexe Datentypen

12.1 Strukturen

*Strukturen*fassen mehrere primitive oder komplexe Variablen zu einer logischen Einheit zusammen. Die Variablen dürfen dabei unterschiedliche Datentypen besitzen. Die Variablen der Struktur werden als *Komponenten*(engl. members) bezeichnet.

Eine logische Einheit kann beispielsweise eine Adresse, Koordinaten, Datums- oder Zeitangaben sein. Ein Datum besteht beispielsweise aus den Komponenten Tag, Monat und Jahr. Eine solche Deklaration einer Struktur sieht dann wie folgt aus:

```
struct datum
{
  int tag;
  char monat[10];
  int jahr;
};
```

Vergessen Sie bei der Deklaration bitte nicht das Semikolon am Ende!

Es gibt mehrere Möglichkeiten, Variablen von diesem Typ zu erzeugen, zum Beispiel

```
struct datum
{
  int tag;
  char monat[10];
  int jahr;
} geburtstag, urlaub;
```

Die zweite Möglichkeit besteht darin, den Strukturtyp zunächst wie oben zu deklarieren, die Variablen von diesem Typ aber erst später zu definieren:

```
struct datum geburtstag, urlaub;
```

Die Größe einer Variable vom Typ *struct datum*kann mit `sizeof(struct datum)`ermittelt werden. Die Gesamtgröße eines *struct*-Typs kann mehr sein als die Größe der einzelnen Komponenten, in unserem Fall also `sizeof(int) + sizeof(char[10]) + sizeof(int)`. Der Compiler darf nämlich die einzelnen Komponenten so im Speicher ausrichten, dass ein schneller Zugriff möglich ist. Beispiel:

```
struct Test
{
 char c;
 int i;
};

sizeof(struct Test); // Ergibt wahrscheinlich nicht 5
```

Der Compiler wird vermutlich 8 oder 16 Byte für die Struktur reservieren.

```
struct Test t;
printf("Addr t   : %p\n", &t);
printf("Addr t.c : %p\n", &t.c);
printf("Addr t.i : %p\n", &t.i);
```

Die Ausgabe wird ungefähr so aussehen.

```
Addr t   : 0x0800
Addr t.c : 0x0800
Addr t.i : 0x0804
```

Die Bytes im Speicher an Adresse 0x801, 0x0802, 0x803 bleiben also ungenutzt. Zu beachten ist außerdem, dass auch wenn *char c*und *int i*vertauscht werden, wahrscheinlich 8 Byte reserviert werden, damit das nächste Element wieder an einer 4-Byte-Grenze ausgerichtet ist.

Die Zuweisung kann komponentenweise erfolgen oder in geschweifter Klammer:

```
struct datum geburtstag = {7, "Mai", 2005};
```

Beim Zugriff auf eine Strukturvariable muss immer der Bezeichner der Struktur durch einen Punkt getrennt mit angegeben werden. Mit

```
geburtstag.jahr = 1964;
```

wird der Komponente `jahr`der Struktur `geburtstag`der neue Wert 1964 zugewiesen.

Der gesamte Inhalt einer Struktur kann einer anderen Struktur zugewiesen werden. Mit

```
urlaub = geburtstag;
```

wird der gesamte Inhalt der Struktur `geburtstag`dem Inhalt der Struktur `urlaub`zugewiesen.

Es gibt auch Zeiger auf Strukturen. Mit

```
struct datum *urlaub;
```

wird *urlaub*als ein Zeiger auf eine Variable vom Typ *struct datum*vereinbart. Der Zugriff auf das Element `tag`erfolgt über `(*urlaub).tag`.

Die Klammern sind nötig, da der Vorrang des Punktoperators höher ist als der des Dereferenzierungsoperators `*`. Würde die Klammer fehlen, würde der Dereferenzierungsoperator auf den gesamten Ausdruck angewendet, so dass man stattdessen `*(urlaub.tag)`erhalten würde. Da die Komponente `tag`aber kein Zeiger ist, würde man hier einen Fehler erhalten.

Da Zeiger auf Strukturen sehr häufig gebraucht werden, wurde in C der `->`-Operator (auch Strukturoperator genannt)eingeführt. Er steht an der Stelle des Punktoperators. So ist beispielsweise `(*urlaub).tag`äquivalent zu `urlaub->tag`.

12.2 Unions

Unions sind Strukturen sehr ähnlich. Der Hauptunterschied zwischen Strukturen und Unions liegt allerdings darin, dass die Elemente denselben Speicherplatz bezeichnen. Deshalb benötigt eine Variable vom Typ union nur genau soviel Speicherplatz, wie ihr jeweils größtes Element.

Unions werden immer da verwendet, wo man komplexe Daten interpretieren will. Zum Beispiel beim Lesen von Datendateien. Man hat sich ein bestimmtes Datenformat ausgedacht, weiß aber erst beim Interpretieren, was man mit den Daten anfängt. Dann kann man mit den Unions alle denkbaren Fälle deklarieren und je nach Kontext auf die Daten zugreifen. Eine andere Anwendung ist die Konvertierung von Daten. Man legt zwei Datentypen "übereinander" und kann auf die einzelnen Teile zugreifen.

Im folgenden Beispiel wird ein `char`-Element mit einem `short`-Element überlagert. Das `char`-Element belegt genau 1 Byte, während das `short`-Element 2 Byte belegt.

Beispiel:

```
union zahl
{
  char  c_zahl; //1 Byte
  short s_zahl; //1 Byte + 1 Byte
}z;
```

Mit

```
z.c_zahl = 5;
```

wird dem Element c_zahl der Variable z der Wert `5`zugwiesen. Da sich `c_zahl`und das erste Byte von `s_zahl`auf derselben Speicheradresse befinden, werden nur die 8 Bit des Elements `c_zahl`verändert. Die nächsten 8 Bit, welche benötigt werden, wenn Daten vom Typ `short`in die Variable z geschrieben werden, bleiben unverändert. Wird nun versucht auf ein Element zuzugreifen, dessen Typ sich vom Typ des Elements unterscheidet, auf das zuletzt geschrieben wurde, ist das Ergebnis nicht immer definiert.

Wie auch bei Strukturen kann der ->Operator auf eine Variable vom Typ Union angewendet werden.

Unions und Strukturen können beinahe beliebig ineinander verschachtelt werden. Eine Union kann also innerhalb einer Struktur definiert werden und umgekehrt.

Beispiel:

```
union vector3d {
 struct { float x, y, z; } vec1;
 struct { float alpha, beta, gamma; } vec2;
 float vec3[3];
};
```

Um den in der Union aktuell verwendeten Datentyp zu erkennen bzw. zu speichern, bietet es sich an, eine Struktur zu definieren, die die verwendete Union zusammen mit einer weiteren Variable umschliesst. Diese weitere Variable kann dann entsprechend kodiert werden, um den verwendeten Typ abzubilden:

```
struct checkedUnion {
 int type;  // Variable zum Speichern des in der Union verwendeten Datentyps
 union intFloat {
  int i;
  float f;
 } intFloat1;
};
```

Wenn man jetzt eine Variable vom Typ `struct checkedUnion`deklariert, kann man bei jedem Lese- bzw. Speicherzugriff den gespeicherten Datentyp abprüfen bzw. ändern. Um nicht direkt mit Zahlenwerten für die verschiedenen Typen zu arbeiten, kann man sich Konstanten definieren, mit denen man dann bequem arbeiten kann. So könnte der Code zum Abfragen und Speichern von Werten aussehen:

```
#include <stdio.h>

#define UNDEF 0
#define INT 1
#define FLOAT 2

int main (void) {

 struct checkedUnion {
  int type;
  union intFloat {
   int i;
   float f;
  } intFloat1;
 };

 struct checkedUnion test1;
 test1.type = UNDEF; // Initialisierung von type mit UNDEF=0, damit der
 undefinierte Fall zu erkennen ist
 int testInt = 10;
 float testFloat = 0.1;

 /* Beispiel für einen Integer */
 test1.type = INT; // setzen des Datentyps für die Union
 test1.intFloat1.i = testInt; // setzen des Wertes der Union

 /* Beispiel für einen Float */
 test1.type = FLOAT;
 test1.intFloat1.f = testFloat;

 /* Beispiel für einen Lesezugriff */
 if (test1.type == INT) {
  printf ("Der Integerwert der Union ist: %d\n", test1.intFloat1.i);
 } else if (test1.type == FLOAT) {
  printf ("Der Floatwert der Union ist: %lf\n", test1.intFloat1.f);
 } else {
  printf ("FEHLER!\n");
 }

 return 0;
}
```

Folgendes wäre also nicht möglich, da die von der Union umschlossene Struktur zwar definiert aber nicht deklariert wurde:

```
union impossible {
 struct { int i, j; char l; }; // Deklaration fehlt, richtig wäre: struct { ...
} structName;
 float b;
```

```
  void* buffer;
};
```

Unions sind wann immer es möglich ist zu vermeiden. Type punning[1] (engl.) – zu deutsch etwa *spielen mit den Datentypen*– ist eine sehr fehlerträchtige Angelegenheit und erschwert das Kompilieren auf anderen und die Interoperabilität mit anderen Systemen mitunter ungemein.

12.3 Aufzählungen

Die Definition eines Aufzählungsdatentyps (*enum*) hat die Form

```
enum [Typname] {
    Bezeichner [= Wert] {, Bezeichner [= Wert]}
};
```

Damit wird der Typ *Typname*definiert. Eine Variable diesen Typs kann einen der mit *Bezeichner*definierten Werte annehmen. Beispiel:

```
enum Farbe {
    Blau, Gelb, Orange, Braun, Schwarz
};
```

Aufzählungstypen sind eigentlich nichts anderes als eine Definition von vielen Konstanten. Durch die Zusammenfassung zu einem Aufzählungstyp wird ausgedrückt, dass die Konstanten miteinander verwandt sind. Ansonsten verhalten sich diese Konstanten ähnlich wie Integerzahlen, und die meisten Compiler stört es auch nicht, wenn man sie bunt durcheinander mischt, also zum Beispiel einer *int*-Variablen den Wert *Schwarz*zuweist.

Für Menschen ist es sehr hilfreich, Bezeichner statt Zahlen zu verwenden. So ist bei der Anweisung *textfarbe(4)*nicht gleich klar, welche Farbe denn zur 4 gehört. Benutzt man jedoch *textfarbe(Schwarz)*, ist der Quelltext leichter lesbar.

Bei der Definition eines Aufzählungstyps wird dem ersten Bezeichner der Wert 0 zugewiesen, falls kein Wert explizit angegeben wird. Jeder weitere Bezeichner erhält den Wert seines Vorgängers, erhöht um 1. Beispiel:

```
enum Primzahl {
    Zwei = 2, Drei, Fuenf = 5, Sieben = 7
};
```

Die *Drei*hat keinen expliziten Wert bekommen. Der Vorgänger hat den Wert 2, daher wird *Drei*$= 2 + 1 = 3$.

Meistens ist es nicht wichtig, welcher Wert zu welchem Bezeichner gehört, Hauptsache sie sind alle unterschiedlich. Wenn man die Werte für die Bezeichner nicht selbst festlegt (so wie im Farbenbeispiel oben), kümmert sich der Compiler darum, dass jeder Bezeichner einen eindeutigen Wert bekommt. Aus diesem Grund sollte man mit dem expliziten Festlegen auch sparsam umgehen.

1 http://en.wikipedia.org/wiki/Type%20punning

12.4 Variablen-Deklaration

Es ist zu beachten, dass z.B. Struktur-Variablen wie folgt deklariert werden müssen:

```
struct StrukturName VariablenName;
```

Dies kann umgangen werden, indem man die Struktur wie folgt definiert:

```
typedef struct
{
  // Struktur-Elemente
} StrukturName;
```

Dann können die Struktur-Variablen einfach durch

```
StrukturName VariablenName;
```

deklariert werden. Dies gilt nicht nur für Strukturen, sondern auch für Unions und Aufzählungen.

Folgendes ist auch möglich, da sowohl der Bezeichner `struct StrukturName`, wie auch `StrukturName`, definiert wird:

```
typedef struct StrukturName
{
   // Struktur-Elemente
} StrukturName;

StrukturName VariablenName1;
struct StrukturName VariablenName2;
```

Mit typedef können Typen erzeugt werden, ähnlich wie ”int” und ”char” welche sind. Dies ist hilfreich um seinen Code noch genauer zu strukturieren.

Beispiel:

```
typedef char name[200];
typedef char postleitzahl[5];

typedef struct {
        name strasse;
        unsigned int hausnummer;
        postleitzahl plz;
} adresse;

int main()
{
        name vorname, nachname;
        adresse meine_adresse;
}
```

en:C Programming/Complex types[2] pl:C/Typy złożone[3]

2 http://en.wikibooks.org/wiki/C%20Programming%2FComplex%20types
3 http://pl.wikibooks.org/wiki/C%2FTypy%20z%C5%82o%C5%BCone

13 Typumwandlung

Der Typ eines Wertes kann sich aus verschiedenen Gründen ändern müssen. Beispielsweise, weil man unter Berücksichtigung höherer Genauigkeit weiter rechnen möchte, oder weil man den Nachkomma-Teil eines Wertes nicht mehr benötigt. In solchen Fällen verwendet man Typumwandlung (auch als Typkonvertierung bezeichnet).

Man unterscheidet dabei grundsätzlich zwischen **expliziter**und **impliziter**Typumwandlung. Explizite Typumwandlung nennt man auch *Cast*.

Eine Typumwandlung kann *einschränkend*oder *erweiternd*sein.

13.1 Implizite Typumwandlung

Bei der impliziten Typumwandlung wird die Umwandlung nicht im Code aufgeführt. Sie wird vom Compiler automatisch anhand der Datentypen von Variablen bzw. Ausdrücken erkannt und durchgeführt. Beispiel:

```
int i = 5;
float f = i; // implizite Typumwandlung
```

Offenbar gibt es hier kein Problem. Unsere Ganzzahl 5 wird in eine Gleitkommazahl umgewandelt. Dabei könnten die ausgegebenen Variablen zum Beispiel so aussehen:

```
5
5.000000
```

Die implizite Typumwandlung (allgemeiner **Erweiternde Typumwandlung**) erfolgt von kleinen zu größeren Datentypen.

13.2 Explizite Typumwandlung

Anders als bei der impliziten Typumwandlung wird die explizite Typumwandlung im Code angegeben. Es gilt folgende Syntax:

```
(Zieltyp)Ausdruck
```

Wobei *Zieltyp*der Datentyp ist, zu dem *Ausdruck*konvertiert werden soll. Beispiel:

```
float pi = 3.14159;
int i = (int)pi; // explizite Typumwandlung
```

liefert i=3.

Die explizite Typumwandlung entspricht allgemein dem Konzept der **Einschränkenden Typumwandlung**.

13.3 Verhalten von Werten bei Typumwandlungen

Fassen wir zusammen. Wandeln wir `int`in `float`um, wird impliziert erweitert, d. h. es geht keine Genauigkeit verloren.

Haben wir eine `float`nach `int`Umwandlung, schneidet der Compiler die Nachkommastellen ab - Genauigkeit geht zwar verloren, aber das Programm ist in seiner Funktion allgemein nicht beeinträchtigt.

Werden allgemein größere in kleinere Ganzzahltypen umgewandelt, werden die oberen Bits abgeschnitten (es erfolgt somit keine Rundung!). Würde man versuchen einen Gleitpunkttyp in einen beliebigen Typ mit kleineren Wertebereich umzuwandeln, ist das Verhalten unbestimmt.

14 Speicherverwaltung

Die Daten, mit denen ein Programm arbeitet, müssen während der Laufzeit an einem bestimmten Ort der Computer-Hardware abgelegt und zugreifbar sein. Die Speicherverwaltung bestimmt, wo bestimmte Daten abgelegt werden, und wer (welche Programme, Programmteile) wie (nur lesen oder auch schreiben) darauf zugreifen darf. Zudem unterscheidet man Speicher auch danach, wann die Zuordnung eines Speicherortes überhaupt stattfindet. Die Speicherverwaltung wird in erster Linie durch die Deklaration einer Variablen (oder Konstanten) beeinflusst, aber auch durch Pragmas und durch Laufzeit-Allozierung, üblicherweise malloc oder calloc.

14.1 Ort und Art der Speicherreservierung (Speicherklasse)

Zum Teil bestimmt der Ort eines Speichers die Zugriffsmöglichkeiten und -geschwindigkeiten, zum Teil wird der Zugriff aber auch von Compiler, Betriebssystem und Hardware kontrolliert.

14.1.1 Speicherorte

Mögliche physikalische Speicherorte sind in erster Linie die Register der CPU und der Arbeitsspeicher.

Um eine Variable explizit in einem Register abzulegen, deklariert man eine Variable unter der Speicherklasse `register`, z.B.:

```
register int var;
```

Von dieser Möglichkeit sollte man allerdings, wenn überhaupt, nur äußerst selten Gebrauch machen, da eine CPU nur wenige Register besitzt, und einige von diesen stets für die Abarbeitung von Maschinenbefehlen benötigt werden. Die meisten Compiler verfügen zudem über Optimierungs-Algorithmen, die Variablen in der Regel dann in Registern ablegen, wenn es am sinnvollsten ist.

Die Ablage im Arbeitsspeicher kann grundsätzlich in zwei verschiedenen Bereichen erfolgen.

Zum einen innerhalb einer Funktion, die Variable hat dann zur Ausführungszeit der Funktion eine Position im Stack oder wird vom Optimierungs-Algorithmus in einem Register platziert. Bei erneutem Aufruf der Funktion hat die Variable dann nicht den gleichen Wert, wie zum Abschluss des letzten Aufrufs. Bei rekursivem Aufruf erhält sie einen neuen, eigenen Speicherplatz, auch mit einem anderen Wert. Deklariert man eine Variable innerhalb einer Funktion ohne weitere Angaben zur Speicherklasse innerhalb eines Funktionskörpers, so gehört sie der Funktion an, z.B:

```
int fun(int var) {
    int var;
}
```

Zum anderen im allgemeinen Bereich des Arbeitsspeichers, außerhalb des Stacks. Dies erreicht man, indem man die Variable entweder außerhalb von Funktionskörpern, oder innerhalb unter der Speicherklasse `static`deklariert:

```
int fun(int var) {
    static int var;
}
```

In Bezug auf Funktionen hat `static`eine andere Bedeutung, siehe ebenda. Ebenfalls im allgemeinen Arbeitsspeicher landen Variablen, deren Speicherort zur Laufzeit alloziert wird, s.u.

Insbesondere bei eingebetteten Systemen gibt es oft unterschiedliche Bereiche des allgemeinen Adressbereichs des Arbeitsspeichers, hauptsächlich unterschieden nach RAM und ROM. Ob eine Variable in direktem Zugriff nur gelesen oder auch geschrieben werden kann, hängt dann also vom Speicherort ab. Der Speicherort einer Variable wird hier durch zusätzliche Compiler-Direktiven, Pragmas, deklariert, deren Syntax sich zwischen den jeweiligen Compilern stark unterscheidet.

14.1.2 Zugriffsverwaltung

14.2 Zeitpunkt der Speicherreservierung

14.2.1 Zum Zeitpunkt des Kompilierens

14.2.2 Zur Ladezeit

14.2.3 Während der Laufzeit

Wenn Speicher für Variablen benötigt wird, z.B. eine skalare Variable mit

```
int var;
```

oder eine Feld-Variable mit

```
int array[10];
```

deklariert werden, wird auch automatisch Speicher auf dem Stack reserviert.

Wenn jedoch die Größe des benötigten Speichers zum Zeitpunkt des Kompilierens noch nicht feststeht, muss der Speicher dynamisch reserviert werden.

Dies geschieht meist mit Hilfe der Funktionen `malloc()`oder `calloc()`aus dem Header stdlib.h[1], der man die Anzahl der benötigten Byte als Parameter übergibt. Die Funktion

1 Kapitel 22.10.13 auf Seite 188

gibt danach einen void-Zeiger auf den reservierten Speicherbereich zurück, den man in den gewünschten Typ casten kann. Die Anzahl der benötigten Bytes für einen Datentyp erhält man mit Hilfe des `sizeof()`-Operators.

Beispiel:

```
int *zeiger;
zeiger = (int *) malloc(sizeof(*zeiger) * 10); /* Reserviert Speicher für 10
 Integer-Variablen
                                        und lässt 'zeiger' auf den
 Speicherbereich zeigen. */
```

Nach dem `malloc()`sollte man testen, ob der Rückgabewert `NULL`ist. Im Erfolgsfall wird `malloc()`einen Wert ungleich NULL zurückgeben. Sollte der Wert aber `NULL`sein ist `malloc()`gescheitert und das System hat nicht genügend Speicher allokieren können. Versucht man, auf diesen Bereich zu schreiben, hat dies ein undefiniertes Verhalten des Systems zur Folge. Folgendes Beispiel zeigt, wie man mit Hilfe einer Abfrage diese Falle umgehen kann:

```
#include <stdlib.h>
#include <stdio.h>
int *zeiger;

zeiger = (int *) malloc(sizeof(*zeiger) * 10);          // Speicher anfordern
if (zeiger == NULL) {
    perror("Nicht genug Speicher vorhanden."); // Fehler ausgeben
    exit(EXIT_FAILURE);                        // Programm mit Fehlercode
 abbrechen
}
free(zeiger);                                  // Speicher wieder freigeben
```

Wenn der Speicher nicht mehr benötigt wird, muss er mit der Funktion `free()`freigegeben werden, indem man als Parameter den Zeiger auf den Speicherbereich übergibt.

```
free(zeiger); // Gibt den Speicher wieder frei
```

Wichtig: Nach dem `free`steht der Speicher nicht mehr zur Verfügung, und jeder Zugriff auf diesen Speicher führt zu undefiniertem Verhalten. Dies gilt auch, wenn man versucht, einen bereits freigegebenen Speicherbereich nochmal freizugeben. Auch ein `free()`auf einen Speicher, der nicht dynamisch verwaltet wird, führt zu einem Fehler. Einzig ein `free()`auf einen NULL-Zeiger ist möglich, da hier der ISO-Standard ISO9899:1999 sagt, dass dieses keine Auswirkungen haben darf. Siehe dazu folgendes Beispiel:

```
int *zeiger;
int *zeiger2;
int *zeiger3;
int array[10];

zeiger = (int *) malloc(sizeof(*zeiger) * 10);  // Speicher anfordern
zeiger2 = zeiger;
zeiger3 = zeiger++;
free(zeiger);                          // geht noch gut
free(zeiger2);                         // FEHLER: DER BEREICH IST SCHON
 FREIGEGEBEN
free(zeiger3);                         /* undefiniertes Verhalten, wenn der
 Bereich
                                          nicht schon freigeben worden wäre. So
 ist
                                          es ein FEHLER
     */
```

```
free(array);                            // FEHLER: KEIN DYNAMISCHER SPEICHER
free(NULL);                             // KEIN FEHLER, ist laut Standard
 erlaubt
```

15 Verkette Listen

Beim Programmieren in C kommt man immer wieder zu Punkten, an denen man feststellt, dass man mit einem Array nicht auskommt. Diese treten zum Beispiel dann ein, wenn man eine unbekannte Anzahl von Elementen verwalten muss. Mit den Mitteln, die wir jetzt kennen, könnte man beispielsweise für eine Anzahl an Elementen Speicher dynamisch anfordern und wenn dieser aufgebraucht ist, einen neuen größeren Speicher anfordern, den alten Inhalt in den neuen Speicher schreiben und dann den alten wieder löschen. Klingt beim ersten Hinsehen ziemlich ineffizient, Speicher allokieren, füllen, neu allokieren, kopieren und freigeben. Also lassen Sie uns überlegen, wie wir das Verfahren optimieren können.

15.0.4 1. Überlegung:

Wir fordern vom System immer nur Platz für ein Element an. Vorteil: Jedes Element hat einen eigenen Speicher und wir können jetzt für neue Elemente einfach einen malloc ausführen. Weiterhin sparen wir uns das Kopieren, da jedes Element von unserem Programm eigenständig behandelt wird. Nachteil: Wir haben viele Zeiger, die jeweils auf ein Element zeigen und wir können immer noch nicht beliebig viele Elemente verwalten.

15.0.5 2. Überlegung:

Jedes Element ist ein komplexer Datentyp, welcher einen Zeiger enthält, der auf ein Element gleichen Typs zeigen kann. Vorteil: wir können jedes Element einzeln allokieren und so die Vorteile der ersten Überlegung nutzen, weiterhin können wir nun in jedem Element den Zeiger auf das nächste Element zeigen lassen, und brauchen in unserem Programm nur einen Zeiger auf das erste Element. Somit ist es möglich, beliebig viele Elemente zur Laufzeit zu verwalten. Nachteil: Wir können nicht einfach ein Element aus der Kette löschen, da sonst **kein**Zeiger mehr auf die nachfolgenden existiert.

15.1 Die einfach verkettete Liste

Die Liste ist das Resultat der beiden Überlegungen, die wir angestellt haben. Eine einfache Art, eine verkettete Liste zu erzeugen, sieht man im folgenden Beispielquelltext:

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

struct element {
    int value; // der Wert des Elements
    struct element *next; // das nächste Element
};
```

```
void printliste(struct element *l) {
    struct element *liste;
    liste=l;
    printf ("%d\n", liste->value);
    while (liste->next != NULL) {
        liste=liste->next;
        printf ("%d\n", liste->value);
    }
}

void append(struct element **lst, int value)
{
    struct element *neuesElement;
    struct element *lst_iter = *lst;

    neuesElement = (struct element*) malloc(sizeof(*neuesElement)); // erzeuge
 ein neues Element
    neuesElement->value = value;
    neuesElement->next = NULL; // Wichtig für das Erkennen des Listenendes

    if ( lst_iter != NULL ) { // sind Elemente vorhanden
        while (lst_iter->next != NULL ) // suche das letzte Element
            lst_iter=lst_iter->next;
        lst_iter->next=neuesElement; // Hänge das Element hinten an
    }
    else // wenn die liste leer ist, bin ich das erste Element
        *lst=neuesElement;
}

int main()
{
    struct element *Liste;
    Liste = NULL; // init. die Liste mit NULL = leere liste
    append(&Liste, 1); // füge neues Element in die Liste ein
    append(&Liste, 3); // füge neues Element in die Liste ein
    append(&Liste, 2); // füge neues Element in die Liste ein

    printliste(Liste); // zeige liste an
    return EXIT_SUCCESS;
}
```

16 Fehlerbehandlung

*Eine*gute Methode, Fehler zu entdecken, ist es, mit dem Präprozessor eine DEBUG-Konstante zu setzen und in den Code detaillierte Meldungen einzubauen. Wenn dann alle Fehler beseitigt sind und das Programm zufriedenstellend läuft, kann man diese Variable wieder entfernen.

Beispiel:

```
#define DEBUG

int main(void){
    #ifdef DEBUG
    führe foo aus (z.B. printf("bin gerade hier\n"); )
    #endif
    bar; }
```

Eine andere Methode besteht darin, **assert ()**zu benutzen.

```
#include <assert.h>
int main (void)
{
 char *p = NULL;
 /* tu was mit p */
 ...
 /* assert beendet das Programm, wenn die Bedingung FALSE ist */
 assert (p != NULL);
 ...
 return 0;
}
```

Das Makro `assert`ist in der Headerdatei `assert.h`definiert. Dieses Makro dient dazu, eine Annahme (englisch: assertion) zu überprüfen. Der Programmierer geht beim Schreiben des Programms davon aus, dass gewisse Annahmen zutreffen (wahr sind). Sein Programm wird nur dann korrekt funktionieren, wenn diese Annahmen zur Laufzeit des Programms auch tatsächlich zutreffen. Liefert eine Überprüfung mittels `assert`den Wert `TRUE`, läuft das Programm normal weiter. Ergibt die Überprüfung hingegen ein `FALSE`, wird das Programm mit einer Fehlermeldung angehalten. Die Fehlermeldung beinhaltet den Text ”assertion failed” zusammen mit dem Namen der Quelltextdatei und der Angabe der Zeilennummer.

17 Präprozessor

Der Präprozessor ist ein mächtiges und gleichzeitig fehleranfälliges Werkzeug, um bestimmte Funktionen auf den Code anzuwenden, bevor er vom Compiler verarbeitet wird.

17.1 Direktiven

Die Anweisungen an den Präprozessor werden als Direktiven bezeichnet. Diese Direktiven stehen in der Form

```
#Direktive Parameter
```

im Code. Sie beginnen mit # und müssen nicht mit einem Semikolon abgeschlossen werden. Eventuell vorkommende Sonderzeichen in den Parametern müssen nicht escaped werden.

17.1.1 #include

Include-Direktiven sind in den Beispielprogrammen bereits vorgekommen. Sie binden die angegebene Datei in die aktuelle Source-Datei ein. Es gibt zwei Arten der #include-Direktive, nämlich

```
#include <Datei.h>
```

und

```
 #include "Datei.h"
```

Die erste Anweisung sucht die Datei im Standard-Includeverzeichnis des Compilers, die zweite Anweisung sucht die Datei zuerst im Verzeichnis, in der sich die aktuelle Sourcedatei befindet; sollte dort keine Datei mit diesem Namen vorhanden sein, sucht sie ebenfalls im Standard-Includeverzeichnis.

17.1.2 #define

Für die #define-Direktive gibt es verschiedene Anweisungen.

Die erste Anwendung besteht im Definieren eines Symbols mit

```
#define SYMBOL
```

wobei *SYMBOL* jeder gültige Bezeichner in C sein kann. Mit den Direktiven `#ifdef` bzw. `#ifndef` kann geprüft werden, ob diese Symbole definiert wurden.

Die zweite Anwendungsmöglichkeit ist das Definieren einer Konstante mit

```
#define KONSTANTE (Wert)
```

wobei *KONSTANTE* wieder jeder gültige Bezeichner sein darf und *Wert* ist der Wert oder Ausdruck durch den *KONSTANTE* ersetzt wird. Insbesondere wenn arithmetische Ausdrücke als Konstante definiert sind, ist die Verwendung einer Klammer sehr ratsam, da ansonsten eine unerwartete Rangfolge der Operatoren auftreten kann.

Die dritte Anwendung ist die Definition eines Makros mit

```
#define MAKRO Ausdruck
```

wobei *MAKRO* der Name des Makros ist und *Ausdruck* die Anweisungen des Makros darstellt.

17.1.3 #undef

Die Direktive #undef löscht ein mit `define` gesetztes Symbol. Syntax:

```
#undef SYMBOL
```

17.1.4 #ifdef

Mit der #ifdef-Direktive kann geprüft werden, ob ein Symbol definiert wurde. Falls nicht, wird der Code nach der Direktive nicht an den Compiler weitergegeben. Eine #ifdef-Direktive muss durch eine #endif-Direktive abgeschlossen werden.

17.1.5 #ifndef

Die #ifndef-Direktive ist das Gegenstück zur #ifdef-Direktive. Sie prüft, ob ein Symbol nicht definiert ist. Sollte es doch sein, wird der Code nach der Direktive nicht an den Compiler weitergegeben. Eine #ifndef-Direktive muss ebenfalls durch eine #endif-Direktive abgeschlossen werden.

17.1.6 #endif

Die #endif-Direktive schließt die vorhergehende #ifdef-, #ifndef-, #if- bzw #elif-Direktive ab. Syntax:

```
#ifdef SYMBOL
// Code, der nicht an den Compiler weitergegeben wird
#endif

#define SYMBOL
#ifndef SYMBOL
```

```
// Wird ebenfalls nicht kompiliert
#endif
#ifdef SYMBOL
// Wird kompiliert
#endif
```

Solche Konstrukte werden häufig verwendet, um Debug-Anweisungen im fertigen Programm von der Übersetzung auszuschließen oder um mehrere, von außen gesteuerte, Übersetzungsvarianten zu ermöglichen.

17.1.7 #error

Die #error-Direktive wird verwendet, um den Kompilierungsvorgang mit einer (optionalen) Fehlermeldung abzubrechen. Syntax:

```
#error Fehlermeldung
```

Die Fehlermeldung muss nicht in Anführungszeichen stehen.

17.1.8 #if

Mit #if kann ähnlich wie mit #ifdef eine bedingte Übersetzung eingeleitet werden, jedoch können hier konstante Ausdrücke ausgewertet werden.

Beispiel:

```
#if (DEBUGLEVEL >= 1)
#  define print1 printf
#else
#  define print1(...) (0)
#endif

#if (DEBUGLEVEL >= 2)
#  define print2 printf
#else
#  define print2(...) (0)
#endif
```

Hier wird abhängig vom Wert der Präprozessorkonstante DEBUGLEVEL definiert, was beim Aufruf von `print2()`oder `print1()`passiert.

Der Präprozessorausdruck innerhalb der Bedingung folgt den gleichen Regeln wie Ausdrücke in C, jedoch muss das Ergebnis zum Übersetzungszeitpunkt bekannt sein.

defined

definedist ein unärer Operator, der in den Ausdrücken der #**if**und #**elif**Direktiven eingesetzt werden kann.

Beispiel:

```
#define FOO
#if defined FOO || defined BAR
```

```
#error "FOO oder BAR ist definiert"
#endif
```

Die genaue Syntax ist

```
defined SYMBOL
```

Ist das Symbol definiert, so liefert der Operator den Wert 1, anderenfalls den Wert 0.

17.1.9 #elif

Ähnlich wie in einem else-if Konstrukt kann mit Hilfe von #**elif**etwas in Abhängigkeit einer früheren Auswahl definiert werden. Der folgende Abschnitt verdeutlicht das.

```
#define BAR
#ifdef FOO
#error "FOO ist definiert"
#elif defined BAR
#error "BAR ist definiert"
#else
#error "hier ist nichts definiert"
#endif
```

Der Compiler würde hier `BAR ist definiert`ausgeben.

17.1.10 #else

Beispiel:

```
#ifdef FOO
#error "FOO ist definiert"
#else
#error "FOO ist nicht definiert"
#endif
```

#**else**dient dazu, allen sonstigen nicht durch #**ifdef**oder #**ifndef**abgefangenen Fälle einen Bereich zu bieten.

17.1.11 #pragma

Bei den `#pragma`Anweisungen handelt es sich um compilerspezifische Erweiterungen der Sprache C. Diese Anweisungen steuern meist die Codegenerierung. Sie sind aber zu sehr von den Möglichkeiten des jeweiligen Compilers abhängig, als dass man hierzu eine allgemeine Aussage treffen kann. Wenn Interesse an diesen Schaltern besteht, sollte man deshalb in die Dokumentation des Compilers sehen oder sekundäre Literatur verwenden, die sich speziell mit diesem Compiler beschäftigt.

en:C Programming/Preprocessor[1] fr:Programmation C/Préprocesseur[2] it:C/Compilatore e precompilatore/Direttive[3] pl:C/Preprocesor[4]

1 http://en.wikibooks.org/wiki/C%20Programming%2FPreprocessor
2 http://fr.wikibooks.org/wiki/Programmation%20C%2FPr%C3%A9processeur
3 http://it.wikibooks.org/wiki/C%2FCompilatore%20e%20precompilatore%2FDirettive
4 http://pl.wikibooks.org/wiki/C%2FPreprocesor

18 Dateien

In diesem Kapitel geht es um das Thema *Dateien*. Aufgrund der einfachen API stellen wir zunächst die Funktionen rund um Streams vor, mit deren Hilfe Dateien geschrieben und gelesen werden können. Anschließend folgt eine kurze Beschreibung der Funktionen rund um Dateideskriptoren.

18.1 Streams

Die Funktion **fopen**dient dazu, einen Datenstrom (Stream) zu öffnen. Datenströme sind Verallgemeinerungen von Dateien. Die Syntax dieser Funktion lautet:

```
FILE *fopen (const char *Pfad, const char *Modus);
```

Der Pfad ist der Dateiname, der Modus darf wie folgt gesetzt werden:

- r - Datei nur zum Lesen öffnen (READ)
- w - Datei nur zum Schreiben öffnen (WRITE), löscht den Inhalt der Datei, wenn sie bereits existiert
- a - Daten an das Ende der Datei anhängen (APPEND), die Datei wird nötigenfalls angelegt
- r+ - Datei zum Lesen und Schreiben öffnen, die Datei muss bereits existieren
- w+ - Datei zum Lesen und Schreiben öffnen, die Datei wird nötigenfalls angelegt
- a+ - Datei zum Lesen und Schreiben öffnen, um Daten an das Ende der Datei anzuhängen, die Datei wird nötigenfalls angelegt

Es gibt noch einen weiteren Modus():

- b - Binärmodus (anzuhängen an die obigen Modi, z.B. ”rb” oder ”w+b”).

Ohne die Angabe von b werden die Daten im sog. Textmodus gelesen und geschrieben, was dazu führt, dass unter bestimmten Systemen bestimmte Zeichen bzw. Zeichenfolgen interpretiert werden. Unter Windows z.B. wird die Zeichenfolge ”\r\n” als Zeilenumbruch übersetzt. Um dieses zu verhindern, muss die Datei im Binärmodus geöffnet werden. Unter Systemen, die keinen Unterschied zwischen Text- und Binärmodus machen (wie zum Beispiel bei Unix, GNU/Linux), hat das b keine Auswirkungen (es wird bei Unix, GNU/Linux immer im Binärmodus geöffnet).

Die Funktion **fopen**gibt **NULL**zurück, wenn der Datenstrom nicht geöffnet werden konnte, ansonsten einen Zeiger vom Typ **FILE**auf den Datenstrom.

Die Funktion **fclose**dient dazu, die mit der Funktion `fopen`geöffneten Datenströme wieder zu schließen. Die Syntax dieser Funktion lautet:

```
int fclose (FILE *datei);
```

Alle nicht geschriebenen Daten des Stromes ***datei**werden gespeichert, alle ungelesenen Eingabepuffer geleert, der automatisch zugewiesene Puffer wird befreit und der Datenstrom ***datei**geschlossen. Der Rückgabewert der Funktion ist **EOF**, falls Fehler aufgetreten sind, ansonsten ist er **0 (Null)**.

18.1.1 Dateien zum Schreiben öffnen

```
#include <stdio.h>
int main (void)
{
  FILE *datei;
  datei = fopen ("testdatei.txt", "w");
  if (datei == NULL)
  {
    printf("Fehler beim oeffnen der Datei.");
    return 1;
  }
  fprintf (datei, "Hallo, Welt\n");
  fclose (datei);
  return 0;
}
```

Der Inhalt der Datei *testdatei.txt*ist nun:

```
  Hallo, Welt
```

Die Funktion **fprintf**funktioniert genauso, wie die schon bekannte Funktion **printf**. Lediglich das erste Argument muss ein Zeiger auf den Dateistrom sein.

18.1.2 Dateien zum Lesen öffnen

Nachdem wir nun etwas in eine Datei hineingeschrieben haben, versuchen wir in unserem zweiten Programm dieses einmal wieder herauszulesen:

```
#include <stdio.h>

int main (void)
{
  FILE *datei;
  char text[100+1];

  datei = fopen ("testdatei.txt", "r");
  if (datei != NULL)
  {
    fscanf (datei, "%100c", text);
    /* String muss mit Nullbyte abgeschlossen sein */
    text[100] = '\0';
    printf ("%s\n", text);
    fclose (datei);
  }
  return 0;
}
```

Die Ausgabe des Programmes ist wie erwartet

```
Hallo, Welt
```

fscanfist das Pendant zu **scanf**.

18.1.3 Positionen innerhalb von Dateien

Stellen wir uns einmal eine Datei vor, die viele Datensätze eines bestimmten Types beinhaltet, z.B. eine Adressdatei. Wollen wir nun die 4. Adresse ausgeben, so ist es praktisch, an den Ort der 4. Adresse innerhalb der Datei zu springen und diesen auszulesen. Um das folgende Beispiel nicht zu lang werden zu lassen, beschränken wir uns auf *Name*und *Postleitzahl*.

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

/* Die Adressen-Datenstruktur */
typedef struct _adresse
{
  char name[100];
  int plz; /* Postleitzahl */
} adresse;

/* Erzeuge ein Adressen-Record */
void mache_adresse (adresse *a, const char *name, const int plz)
{
  strncpy (a->name, name, 100);
  a->plz = plz;
}

int main (void)
{
  FILE *datei;
  adresse addr;

  /* Datei erzeugen im Binärmodus, ansonsten kann es Probleme
     unter Windows geben, siehe Anmerkungen bei '''fopen()''' */
  datei = fopen ("testdatei.dat", "wb");
  if (datei != NULL)
    {
      mache_adresse (&addr, "Erika Mustermann", 12345);
      fwrite (&addr, sizeof (adresse), 1, datei);
      mache_adresse (&addr, "Hans Müller", 54321);
      fwrite (&addr, sizeof (adresse), 1, datei);
      mache_adresse (&addr, "Secret Services", 700);
      fwrite (&addr, sizeof (adresse), 1, datei);
      mache_adresse (&addr, "Peter Mustermann", 12345);
      fwrite (&addr, sizeof (adresse), 1, datei);
      mache_adresse (&addr, "Wikibook Nutzer", 99999);
      fwrite (&addr, sizeof (adresse), 1, datei);
      fclose (datei);
    }

  /* Datei zum Lesen öffnen - Binärmodus */
  datei = fopen ("testdatei.dat", "rb");
  if (datei != NULL)
    {
      /* Hole den 4. Datensatz */
      fseek(datei, 3 * sizeof (adresse), SEEK_SET);
      fread (&addr, sizeof (adresse), 1, datei);
      printf ("Name: %s (%d)\n", addr.name, addr.plz);
```

```
      fclose (datei);
    }
  return 0;
}
```

Um einen Datensatz zu speichern bzw. zu lesen, bedienen wir uns der Funktionen **fwrite**und **fread**, welche die folgende Syntax haben:

```
size_t fread  (void *daten, size_t groesse, size_t anzahl, FILE *datei);
size_t fwrite (const void *daten, size_t groesse, size_t anzahl, FILE *datei);
```

Beide Funktionen geben die Anzahl der geschriebenen / gelesenen Zeichen zurück. Die *groesse*ist jeweils die Größe eines einzelnen Datensatzes. Es können *anzahl*Datensätze auf einmal geschrieben werden. Beachten Sie, dass sich der Zeiger auf den Dateistrom bei beiden Funktionen am Ende der Argumentenliste befindet.

Um nun an den 4. Datensatz zu gelangen, benutzen wir die Funktion **fseek**:

```
int fseek (FILE *datei, long offset, int von_wo);
```

Diese Funktion gibt *0*zurück, wenn es zu keinem Fehler kommt. Der Offset ist der Ort, dessen Position angefahren werden soll. Diese Position kann mit dem Parameter *von_ wo*beeinflusst werden:

- SEEK_SET - Positioniere relativ zum Dateianfang,
- SEEK_CUR - Positioniere relativ zur aktuellen Dateiposition und
- SEEK_END - Positioniere relativ zum Dateiende.

Man sollte jedoch beachten: wenn man mit dieser Funktion eine Position in einem Textstrom anfahren will, so muss man als Offset *0*oder einen Rückgabewert der Funktion `ftell`angeben (in diesem Fall muss der Wert von *von_ wo*SEEK_SET sein).

18.1.4 Besondere Streams

Neben den Streams, die Sie selbst erzeugen können, gibt es schon vordefinierte:

- **stdin**- Die Standardeingabe (typischerweise die Tastatur)
- **stdout**- Standardausgabe (typischerweise der Bildschirm)
- **stderr**- Standardfehlerkanal (typischerweise ebenfalls Bildschirm)

Diese Streams brauchen nicht geöffnet oder geschlossen zu werden. Sie sind ”einfach schon da”.

```
...
fprintf (stderr, "Fehler: Etwas schlimmes ist passiert\n");
...
```

Wir hätten also auch unsere obigen Beispiele statt mit **printf**mit **fprintf**schreiben können.

18.2 Echte Dateien

Mit ”echten Dateien” bezeichnen wir die API rund um Dateideskriptoren. Hier passiert ein physischer Zugriff auf Geräte. Diese API eignet sich auch dazu, Informationen über angeschlossene Netzwerke zu übermitteln.

18.2.1 Dateiausdruck

Das folgende Beispiel erzeugt eine Datei und gibt anschließend den Dateiinhalt *oktal*, *dezimal*, *hexadezimal*und *als Zeichen*wieder aus. Es soll Ihnen einen Überblick verschaffen über die typischen Dateioperationen: öffnen, lesen, schreiben und schließen.

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <string.h>

int main (void)
{
  int fd;
  char ret;
  const char *s = "Test-Text 0123\n";

  /* Zum Schreiben öffnen */
  fd = open ("testfile.txt", O_WRONLY|O_CREAT|O_TRUNC, S_IRUSR|S_IWUSR);
  if (fd == -1)
    exit (-1);
  write (fd, s, strlen (s));
  close (fd);

  /* Zum Lesen öffnen */
  fd = open ("testfile.txt", O_RDONLY);
  if (fd == -1)
     exit (-1);

  printf ("Oktal\tDezimal\tHexadezimal\tZeichen\n");
  while (read (fd, &ret, sizeof (char)) > 0)
    printf ("%o\t%u\t%x\t\t%c\n", ret, ret, ret, ret);
  close (fd);

 return 0;
}
```

Die Ausgabe des Programms ist wie folgt:

```
  Oktal   Dezimal Hexadezimal     Zeichen
  124     84      54                    T
  145     101     65                    e
  163     115     73                    s
  164     116     74                    t
  55      45      2d                    -
  124     84      54                    T
  145     101     65                    e
  170     120     78                    x
  164     116     74                    t
  40      32      20
```

```
60      48      30                  0
61      49      31                  1
62      50      32                  2
63      51      33                  3
12      10      a
```

Mit **open**erzeugen (**O_CREAT**) wir zuerst eine Datei zum Schreiben (**O_WRONLY**). Wenn diese Datei schon existiert, so soll sie geleert werden (**O_TRUNC**). Derjenige Benutzer, der diese Datei anlegt, soll sie lesen (**S_IRUSR**) und beschreiben (**S_IWUSR**) dürfen. Der Rückgabewert dieser Funktion ist der Dateideskriptor, eine positive ganze Zahl, wenn das Öffnen erfolgreich war. Sonst ist der Rückgabewert *-1*.

In diese so erzeugte Datei können wir schreiben:

```
ssize_t write (int dateideskriptor, const void *buffer, size_t groesse);
```

Diese Funktion gibt die Anzahl der geschriebenen Zeichen zurück. Sie erwartet den Dateideskriptor, einen Zeiger auf einen zu schreibenden Speicherbereich und die Anzahl der zu schreibenden Zeichen.

Der zweite Aufruf von **open**öffnet die Datei zum Lesen (**O_RDONLY**). Bitte beachten Sie, dass der dritte Parameter der **open**-Funktion hier weggelassen werden darf.

Die Funktion **read**erledigt für uns das Lesen:

```
ssize_t read (int dateideskriptor, void *buffer, size_t groesse);
```

Die Parameter sind dieselben wie bei der Funktion **write**. **read**gibt die Anzahl der gelesenen Zeichen zurück.

18.3 Streams und Dateien

In einigen Fällen kommt es vor, dass man - was im allgemeinen keine gute Idee ist - die API der Dateideskriptoren mit der von Streams mischen muss. Hierzu dient die Funktion:

```
FILE *fdopen (int dateideskriptor, const char * Modus);
```

fdopenöffnet eine Datei als Stream, sofern ihr Dateideskriptor vorliegt und der Modus zu den bei **open**angegebenen Modi kompatibel ist.

19 Rekursion

19.1 Rekursion

Eine Funktion, die sich selbst aufruft, wird als rekursive Funktion bezeichnet. Den Aufruf selbst nennt man Rekursion. Als Beispiel dient die Fakultäts-Funktion[1] *n!*, die sich rekursiv als *n(n-1)!*definieren lässt (wobei *0! = 1*).

Hier ein Beispiel dazu in C:

```
#include <stdio.h>

int fakultaet (int a)
{
  if (a == 0)
    return 1;
  else
    return (a * fakultaet(a-1));
}

int main()
{
  int eingabe;

  printf("Ganze Zahl eingeben: ");
  scanf("%d",&eingabe);
  printf("Fakultaet der Zahl: %d\n",fakultaet(eingabe));

  return 0;
}
```

19.2 Beseitigung der Rekursion

Rekursive Funktionen sind in der Regel leichter lesbar als ihre iterativen Gegenstücke. Sie haben aber den Nachteil, dass für jeden Funktionsaufruf verhältnismäßig hohe Kosten anfallen. Eine effiziente Programmierung in C erfordert also die Beseitigung jeglicher Rekursion. Am oben gewählten Beispiel der Fakultät könnte eine rekursionsfreie Variante wie folgt definiert werden:

```
int fak_iter(int n)
{
  int i, fak;
  for (i=1, fak=1; i<=n; i++)
    fak *= i;
```

1 http://de.wikipedia.org/wiki/Fakult%C3%A4t%20%28Mathematik%29

```
  return fak;
}
```

Diese Funktion liefert genau die gleichen Ergebnisse wie die obige, allerdings wurde die Rekursion durch eine Iteration ersetzt. Offensichtlich kommt es innerhalb der Funktion zu keinem weiteren Aufruf, was die Laufzeit des Algorithmus erheblich verkürzen sollte. Komplexere Algorithmen - etwa Quicksort - können nicht so einfach iterativ implementiert werden. Das liegt an der Art der Rekursion, die es bei Quicksort notwendig macht, einen Stack für die Zwischenergebnisse zu verwenden. Eine so optimierte Variante kann allerdings zu einer Laufzeitverbesserung von 25-30% führen.

19.3 Weitere Beispiele für Rekursion

Die Potenzfunktion[2] "y = x hoch n" soll berechnet werden:

```
#include <stdio.h>

int potenz(int x, int n)
{
  if (n>0)
    return (x*potenz(x,--n));  /* rekursiver Aufruf */
  else
    return (1);
}

int main(void)
{
  int x;
  int n;
  int wert;

  printf("\nGib x ein: ");
  scanf("%d",&x);
  printf("\nGib n ein: ");
  scanf("%d",&n);

  if(n<0)
  {
    printf("Exponent muss positiv sein!\n");
    return 1;
  }
  else
  {
    wert=potenz(x,n);
    printf("Funktionswert: %d\n",wert);
    return 0;
  }
}
```

Multiplizieren von zwei Zahlen als Ausschnitt:

```
int multiply(int a, int b)
{
  if (b==0) return 0;
  return a + multiply(a,b-1);
}
```

2 http://de.wikipedia.org/wiki/Potenz_%28Mathematik%29

20 Programmierstil

Ein gewisser Programmierstil ist notwendig, um anderen Programmierern das Lesen des Quelltextes nicht unnötig zu erschweren und um seinen eigenen Code auch nach langer Zeit noch zu verstehen.

Außerdem zwingt man sich durch einen gewissen Stil selbst zum sauberen Programmieren, was die Wartung des Codes vereinfacht.

20.1 Kommentare

Grundsätzlich sollten alle Stellen im Code, die nicht selbsterklärend sind, bestimmtes Vorwissen erfordern oder für andere Stellen im Quelltext kritisch sind, kommentiert werden. Kommentare sollten sich jedoch nur darauf beschränken, zu erklären, WAS eine Funktion macht, und NICHT WIE es gemacht wird.

Eine gute Regel lautet: *Kann man die Funktionalität mit Hilfe des Quelltextes klar formulieren so sollte man es auch tun, ansonsten muss es mit einem Kommentar erklärt werden.* Im englischen lautet die Regel: *If you can say it with code, code it, else comment.*

20.2 Globale Variablen

Globale Variablen sollten vermieden werden, da sie ein Programm sehr anfällig für Fehler machen und schnell zum unsauberen Programmieren verleiten.

Wird eine Variable von mehreren Funktionen innerhalb derselben Datei verwendet, ist es hilfreich, diese Variable als static zu markieren, so dass sie nicht im globalen Namensraum auftaucht.

20.3 Goto-Anweisungen

Die `goto`-Anweisung ist unter Programmierern verpönt, weil sie ein Programm schlecht lesbar machen kann, denn sie kann den Programmlauf völlig zusammenhanglos an jede beliebige Stelle im Programm verzweigen, und so genannten Spaghetti-Code entstehen lassen.

Sie lässt sich fast immer durch Verwenden von Funktionen und Kontrollstrukturen, man nennt dies strukturiertes Programmieren, vermeiden. Es gibt dennoch Fälle, wie z.B. das Exception-Handling mit errno, welche mit Hilfe von goto-Anweisungen leichter realisierbar und sauberer sind.

Generell sollte für saubere Programmierung zumindest gelten, dass eine goto-Verzweigung niemals außerhalb der aktuellen Funktion erfolgen darf. Außerdem sollte hinter der Sprungmarke eines gotos kein weiteres goto folgen.

20.4 Namensgebung

Es gibt viele verschiedene Wege, die man bei der Namensgebung von Variablen, Konstanten, Funktionen usw. beschreiten kann. Zu beachten ist jedenfalls, dass man, egal welches System man verwendet (z.B. Variablen immer klein schreiben und ihnen den Typ als Abkürzung voranstellen und Funktionen mit Großbuchstaben beginnen und zwei Wörter mit Großbuchstaben trennen oder den Unterstrich verwenden), konsequent bleibt. Bei der Sprache, die man für die Bezeichnungen wählt, sei aber etwas angemerkt. Wenn man Open-Source programmieren will, so bietet es sich meist eher an, englische Bezeichnungen zu wählen; ist man aber in einem Team von deutschsprachigen Entwicklern, so wäre wohl die Muttersprache die bessere Wahl. Aber auch hier gilt: Egal was man wählt, man sollte nach der Entscheidung konsequent bleiben.

Da sich alle globalen Funktionen und Variablen einen Namensraum teilen, macht es Sinn, etwa durch Voranstellen des Modulnamens vor den Symbolnamen Eindeutigkeit sicherzustellen. In vielen Fällen lassen sich globale Symbole auch vermeiden, wenn man stattdessen statische Symbole verwendet.

Es sei jedoch angemerkt, dass es meistens nicht sinnvoll ist, Variablen mit nur einem Buchstaben zu verwenden. Es sei denn, es hat sich dieser Buchstabe bereits als Bezeichner in einem Bereich etabliert. Ein Beispiel dafür ist die Variable i als Schleifenzähler oder e, wenn die Eulersche Zahl gebraucht wird. Code ist sehr schlecht zu warten wenn man erstmal suchen muss, welchen Sinn z.B. a hat.

Verbreitete Bezeichner sind:

h, i, j

Laufvariablen in Schleifen

w, x, y, z

Zeilen, Spalten, usw. einer Matrix

r, s, t

Zeiger auf Zeichenketten

20.5 Gestaltung des Codes

Verschiedene Menschen gestalten ihren Code unterschiedlich. Die Einen bevorzugen z.B. bei einer Funktion folgendes Aussehen:

```
int funk(int a){
    return 2 * a;
}
```

andere wiederum würden diese Funktion eher

```
int funk (int a)
{
  return 2 * a;
}
```

schreiben. Es gibt vermutlich so viele unterschiedliche Schreibweisen von Programmen, wie es programmierende Menschen gibt und sicher ist der Eine oder Andere etwas religiös gegenüber der Platzierung einzelner Leerzeichen. Innerhalb von Teams haben sich besondere Vorlieben herauskristallisiert, wie Code auszusehen hat. Um zwischen verschiedenen Gestaltungen des Codes wechseln zu können, gibt es Quelltextformatierer, wie z.B.: GNU indent[1], Artistic Style[2] und eine grafische Oberfläche UniversalIndentGUI[3], die sie bequem benutzen lässt.

20.6 Standard-Funktionen und System-Erweiterungen

Sollte man beim Lösen eines Problem nicht allein mit dem auskommen, was durch den C-Standard erreicht werden kann, ist es sinnvoll die systemspezifischen Teile des Codes in eigene Funktionen und Header [4] zu packen. Dieses macht es leichter den Code auf einem anderen System zu reimplementieren, weil nur die Funktionalität im systemspezifischen Code ausgetauscht werden muss.

1 http://www.gnu.org/software/indent/
2 http://astyle.sourceforge.net/
3 http://universalindent.sourceforge.net/
4 Kapitel 8 auf Seite 75

21 Sicherheit

Wenn man einmal die Grundlagen der C-Programmierung verstanden hat, sollte man mal eine kleine Pause machen. Denn an diesen Punkt werden Sie sicher ihre ersten Programme schreiben wollen, die nicht nur dem Erlernen der Sprache C dienen, sondern Sie wollen für sich und vielleicht auch andere Werkzeuge erstellen, mit denen sich die Arbeit erleichtern lässt. Doch Vorsicht, bis jetzt wurden die Programme von ihnen immer nur so genutzt, wie Sie es dachten.

Wenn Sie so genannten Produktivcode schreiben wollen, sollten Sie davon ausgehen, dass dies nicht länger der Fall sein wird. Es wird immer mal einen Benutzer geben, der nicht das eingibt, was Sie dachten oder der versucht, eine längere Zeichenkette zu verarbeiten, als Sie es bei ihrer Überlegung angenommen haben. Deshalb sollten Sie spätestens jetzt ihr Programm durch eine Reihe von Verhaltensmustern schützen, so gut es geht.

21.1 Sichern Sie Ihr Programm von Anfang an

Einer der Hauptfehler beim Programmieren ist es, zu glauben, erst muss das Programm laufen, dann wird es abgesichert. **Vergessen Sie es!**Wenn es endlich läuft, hängen Sie schon längst im nächsten Projekt. Dann nochmal aufräumen, das macht keiner. Also schreiben Sie vom Beginn an ein sicheres Programm.

21.2 Die Variablen beim Programmstart

Wenn ein Programm gestartet wird, sind erstmal alle Variablen undefiniert. Das heißt, sie haben irgendwelche quasi Zufallswerte. Also weisen Sie jeder Variablen einen Anfangswert zu, auch wenn zufällig der von Ihnen benutzte Compiler die Variablen zum Beginn auf 0 setzt, wie das einige wenige machen. Der nächste Compiler **wird**es anders machen, und Sie stehen dann auf einmal haareraufend vor einem Programm, was eigentlich bis jetzt immer gelaufen ist.

21.3 Der Compiler ist dein Freund

Viele ignorieren die Warnungen, die der Compiler ausgibt, oder haben sie gar nicht angeschaltet. Frei nach dem Motto ”solange es kein Fehler ist”. Dies ist mehr als kurzsichtig. Mit Warnungen will der Compiler uns mitteilen, dass wir gerade auf dem Weg in die Katastrophe sind. Also gleich von Beginn an den Warnungen nachgehen und dafür sorgen, dass diese nicht mehr erscheinen. Wenn sich die Warnungen in einem ganz speziellen Fall nicht

beseitigen lassen, ist es selbstverständlich, dass man dem Projekt eine Erklärung beilegt, die ganz genau erklärt, woher die Warnung kommt, warum man diese nicht umgehen kann und es ist zu beweisen, dass die Warnung unter keinen Umständen zu einem Programmversagen führen wird. Also im Klartext: "Ist halt so" ist keine Begründung.

Wenn Sie ihre Programme mit dem GNU C Compiler schreiben, sollten Sie dem Compiler mindestens diese Argumente mitgeben, um viele sinnvolle Warnungen zu sehen:

```
gcc -Wall -W -Wstrict-prototypes -O
```

Auch viele andere Compiler können sinnvolle Warnungen ausgeben, wenn Sie ihnen die entsprechenden Argumente mitgeben.

21.4 Zeiger und der Speicher

Zeiger sind in C ohne Zweifel eine mächtige Waffe, aber Achtung! Es gibt eine Menge Programme, bei denen es zu sogenannten Pufferüberläufen (Buffer Overflows) [1] gekommen ist, weil der Programmierer sich nicht der Gefahr von Zeigern bewusst war. Wenn Sie also mit Zeigern hantieren, nutzen Sie die Kontrollmöglichkeiten. *malloc()*oder *fopen()*geben im Fehlerfall z.B. *NULL*zurück. Testen Sie also, ob das Ergebis NULL ist und/oder nutzen Sie andere Kontrollen, um zu überprüfen, ob Ihre Zeiger auf gültige Inhalte zeigen.

21.5 Strings in C

Wie Sie vielleicht wissen, sind Strings in C nichts anderes als ein Array von char. Das hat zur Konsequenz, dass es bei Stringoperationen besonders oft zu Pufferüberläufen kommt, weil der Programmierer einfach nicht mit überlangen Strings gerechnet hat. Vermeiden Sie dies, indem Sie nur die Funktionen verwenden, welche die Länge des Zielstrings überwachen:

- `snprintf`statt `sprintf`
- `strncpy`statt `strcpy`

Lesen Sie sich unbedingt die Dokumentation durch, die zusammen mit diesen Funktionen ausgeliefert wird. *strncpy*ist zwar etwas sicherer als *strcpy*, aber es ist trotzdem nicht leicht, sie richtig zu benutzen. Überlegen Sie sich auch, was im Falle von zu langen Strings passieren soll. Falls der String nämlich später benutzt wird, um eine Datei zu löschen, könnte es leicht passieren, dass eine falsche Datei gelöscht wird.

21.6 Das Problem der Reellen Zahlen (Floating Points)

Auch wenn es im C-Standard die Typen "float" und "double" gibt, so sind diese nur bedingt einsatzfähig. Durch die interne Darstellung einer Floatingpointzahl auf eine fest definierte

1 `http://de.wikipedia.org/wiki/Buffer_Overflow`

Anzahl von Bytes in Exponentialschreibweise, kann es bei diesen Datentypen schnell zu Rundungsfehlern kommen. Deshalb sollten Sie in ihren Projekten überlegen ob Sie nicht die Float-Berechnungen durch Integerdatentypen ersetzen können, um eine bessere Genauigkeit zu erhalten. So kann beispielsweise bei finanzmathematischen Programmen, welche cent- oder zehntelcentgenau rechnen, oft der Datentyp "int" benutzt werden. Erst bei der Ausgabe in Euro wird wieder in die Fließkommadarstellung konvertiert.

21.7 Die Eingabe von Werten

Falls Sie eine Eingabe erwarten, gehen Sie immer vom Schlimmsten aus. Vermeiden Sie, einen Wert vom Benutzer ohne Überprüfung zu verwenden. Denn wenn Sie zum Beispiel eine Zahl erwarten, und der Benutzer gibt einen Buchstaben ein, sind meist Ihre daraus folgenden Berechnungen Blödsinn. Also besser erst als Zeichenkette einlesen, dann auf Gültigkeit prüfen und erst dann in den benötigen Typ umwandeln. Auch das Lesen von Strings sollten Sie überdenken: Zum Beispiel prüft der folgende Aufruf die Länge **nicht**!

```
scanf("%s",str);
```

Wenn jetzt der Bereich *str*nicht lang genug für die Eingabe ist, haben Sie einen Pufferüberlauf. Abhilfe schafft hier die Verwendung der Funktion *fgets*:

```
char str[10];
fgets(str, 10, stdin);
```

Hier muss im 2. Parameter angegeben werden, wie groß der verwendete Puffer ist. Wenn hier 10 angegeben ist, werden maximal 9 Zeichen eingelesen, da am Ende noch das Null-Zeichen angehängt werden muss. Besonderheiten dieser Funktion sind, dass sie, sofern der Puffer dafür ausreicht, eine komplette Zeile bis zum Zeilenumbruch einliest. Auch Leerzeichen etc. werden eingelesen und der Zeilenumbruch ist danach ebenfalls im String enthalten.

Meistens hat das jeweilige System noch ein paar Nicht-Standard-Eingabefunktionen, die es besser ermöglichen, die Eingabe zu überprüfen als z. B. *scanf*& co.

21.8 Magic Numbers sind böse

Wenn Sie ein Programm schreiben und dort Berechnungen anstellen oder Register setzen, sollten Sie es vermeiden, dort direkt mit Zahlen zu arbeiten. Nutzen Sie besser die Möglichkeiten von Defines oder Konstanten, die mit sinnvollen Namen ausgestattet sind. Denn nach ein paar Monaten können selbst Sie nicht mehr sagen, was die Zahl in Ihrer Formel sollte. Hierzu ein kleines Beispiel:

```
x=z*9.81;    // schlecht: man kann vielleicht ahnen was der Programmierer will
F=m*9.81;    /* besser: wir können jetzt an der Formel vielleicht schon
                        erkennen: es geht um Kraftberechnung */
#define GRAVITY 9.81
F=m*GRAVITY; // am besten: jeder kann jetzt sofort sagen worum es geht
```

Auch wenn Sie Register haben, die mit ihren Bits irgendwelche Hardware steuern, sollten Sie statt den Magic Numbers einfach einen Header [2] schreiben, welcher über defines den einzelnen Bits eine Bedeutung gibt, und dann über das binäre ODER eine Maske schaffen die ihre Ansteuerung enthält, hierzu ein Beispiel:

```
counters= 0x74;  // Schlecht
counters= COUNTER1 | BIN_COUNTER | COUNTDOWN | RATE_GEN ; // Besser
```

Beide Zeilen machen auf einem fiktiven Mikrocontroller das gleiche, aber für den Code in Zeile 1 müsste ein Programmierer erstmal die Dokumentation des Projekts, wahrscheinlich sogar die des Mikrocontroller lesen, um die Zählrichtung zu ändern. In der Zeile 2 weiß jeder, dass das COUNTDOWN geändert werden muss, und wenn der Entwickler des Headers gut gearbeitet hat, ist auch ein COUNTUP bereits definiert.

21.9 Die Zufallszahlen

„Gott würfelt nicht“ soll Einstein gesagt haben; vielleicht hatte er recht, aber sicher ist, der Computer würfelt auch nicht. Ein Computer erzeugt Zufallszahlen, indem ein Algorithmus Zahlen ausrechnet, die - mehr oder weniger - zufällig verteilt (d.h. zufällig groß) sind. Diese nennt man Pseudozufallszahlen. Die Funktion `rand()`aus der stdlib.h ist ein Beispiel dafür. Für einfache Anwendungen mag `rand()`ausreichen, allerdings ist der verwendete Algorithmus nicht besonders gut, so dass die hiermit erzeugten Zufallszahlen einige schlechte statistische Eigenschaften aufweisen. Eine Anwendung ist etwa in Kryptografie oder Monte-Carlo-Simulationen nicht vertretbar. Hier sollten bessere Zufallszahlengeneratoren eingesetzt werden. Passende Algorithmen finden sich in der *GNU scientific library*`http://www.gnu.org/software/gsl/` oder in *Numerical Recipes*`http://nr.com` (C Version frei zugänglich `http://www.nrbook.com/a/bookcpdf.php`).

21.10 Undefiniertes Verhalten

Es gibt einige Funktionen, die in gewissen Situationen ein undefiniertes Verhalten an den Tag legen. Das heißt, Sie wissen in der Praxis dann nicht, was passieren wird: Es kann passieren, dass das Programm bis in alle Ewigkeit läuft – oder auch nicht. Meiden Sie undefiniertes Verhalten! Sie begeben sich sonst in die Hand des Compilers und was dieser daraus macht. Auch ein ”bei mir läuft das aber” ist keine Erlaubnis, mit diesen Schmutzeffekten zu arbeiten. Das undefinierte Verhalten zu nutzen grenzt an Sabotage.

21.11 return-Statement fehlt

Wenn für eine Funktion zwar ein Rückgabewert angegeben wurde, jedoch ohne return-Statement endet, gibt der Compiler bei Standardeinstellung keinen Fehler aus. Problematisch an diesem Zustand ist, dass eine solche Funktion in diesem Fall eine zufällige, nicht

2 Kapitel 8 auf Seite 75

festgelegte Zahl zurück gibt. Abhilfe schafft nur ein höheres Warning-Level (siehe #Der Compiler ist dein Freund[3]]) bzw. explizit diese Warnungen mit dem Parameter `-Wreturn-type`einzuschalten.

21.12 Wartung des Codes

Ein Programm ist ein technisches Produkt, und wie alle anderen technischen Produkte sollte es wartungsfreundlich sein. So dass Sie oder Ihr Nachfolger in der Lage sind, sich schnell wieder in das Progamm einzuarbeiten. Um das zu erreichen, sollten Sie sich einen einfach zu verstehenden Programmierstil[4] für das Projekt suchen und sich selbst dann an den Stil halten, wenn ein anderer ihn verbrochen hat. Beim Linux-Kernel werden auch gute Patches abgelehnt, weil sie sich z.B. nicht an die Einrücktiefe gehalten haben.

21.13 Wartung der Kommentare

Auch wenn es trivial erscheinen mag, wenn Sie ein Quellcode ändern, vergessen Sie nicht den Kommentar. Man könnte argumentieren, dass der Kommentar ein Teil Ihres Programms ist und so auch einer Wartung unterzogen werden sollte, wie der Code selbst. Aber die Wahrheit ist eigentlich viel einfacher; ein Kommentar, der von der Programmierung abweicht, sorgt bei dem Nächsten, der das Programm ändern muss, erstmal für große Fragezeichen im Kopf. Denn wie wir im Kapitel Programmierstil[5] besprochen haben, soll der Kommentar helfen, die Inhalte des so genannten Fachkonzeptes zu verstehen und dieser Prozess dauert dann viel länger, als mit den richtigen Kommentaren.

21.14 Weitere Informationen

Ausführlich werden die Fallstricke in C und die dadurch möglichen Sicherheitsprobleme im *CERT C Secure Coding Standard*dargestellt `https://www.securecoding.cert.org/confluence/display/seccode/CERT+C+Secure+Coding+Standard`. Er besteht aus einem Satz von Regeln und Empfehlungen, die bei der Programmierung beachtet werden sollten.

3 Kapitel 21.3 auf Seite 151
4 Kapitel 20 auf Seite 147
5 Kapitel 20 auf Seite 147

22 Referenzen

Um C-Programme ausführen zu können, müssen diese erst in die Maschinensprache übersetzt werden. Diesen Vorgang nennt man kompilieren.

Anschließend wird der beim Kompilieren entstandene Objektcode mit einem Linker gelinkt, so dass alle eingebundenen Bibliotheksfunktionen verfügbar sind. Das gelinkte Produkt aus einer oder verschiedenen Objektcode-Dateien und den Bibliotheken ist dann das ausführbare Programm.

22.1 Compiler

Um die erstellten Code-Dateien zu kompilieren, benötigt man selbstverständlich auch einen Compiler. Je nach Plattform hat man verschiedene Alternativen:

22.1.1 Microsoft Windows

Wer zu Anfang nicht all zu viel Aufwand betreiben will, kann mit relativ kleinen Compilern (ca. 2-5 MByte) inkl. IDE/Editor anfangen:

- Pelles C[1], kostenlos. Hier[2] befindet sich der deutsche Mirror.
- lcc-win32[3], kostenlos für private Zwecke.
- cc386[4], Open Source.

Wer etwas mehr Aufwand (finanziell oder an Download) nicht scheut, kann zu größeren Paketen inkl. IDE greifen:

- Microsoft Visual Studio[5], kommerziell, enthält neben dem C-Compiler auch Compiler für C#, C++ und VisualBasic. Visual C++ Express[6] ist die kostenlose Version.
- CodeGear C++ Builder[7], kommerziell, ehemals Borland C++ Builder.
- Open Watcom[8], Open Source.
- wxDevCpp[9], komplette IDE basierend auf dem GNU C Compiler (Mingw32), Open Source.

1 http://www.smorgasbordet.com/pellesc/
2 http://www.pellesc.de/
3 http://www.cs.virginia.edu/~lcc-win32/
4 http://www.members.tripod.com/~ladsoft/cc386.htm
5 http://www.microsoft.com/germany/VisualStudio/
6 http://www.microsoft.com/germany/express/
7 http://www.codegear.com/products/cppbuilder
8 http://www.openwatcom.org/
9 http://wxdsgn.sourceforge.net/

Wer einen (kostenlosen) Kommandozeilen-Compiler bevorzugt, kann zusätzlich zu obigen noch auf folgende Compiler zugreifen:

- Mingw32[10], der GNU-Compiler für Windows, Open Source.
- Digital Mars Compiler[11], kostenlos für private Zwecke.
- Version 5.5 des Borland Compilers[12], kostenlos für private Zwecke (Konfiguration[13] und Gebrauch[14]).

22.1.2 Unix und Linux

Für alle Unix Systeme existieren C-Compiler, die meist auch schon vorinstalliert sind. Insbesondere, bzw. darüber hinaus, existieren folgende Compiler:

- GNU C Compiler[15], Open Source. Ist Teil jeder Linux-Distribution, und für praktisch alle Unix-Systeme verfügbar.
- clang[16], Open Source.
- Tiny C Compiler[17], Open Source.
- Portable C Compiler[18], Open Source.
- Der Intel C/C++ Compiler[19], kostenlos für private Zwecke.

Alle gängigen Linux-Distributionen stellen außerdem zahlreiche Entwicklungsumgebungen zur Verfügung, die vor allem auf den GNU C Compiler zurückgreifen.

22.1.3 Macintosh

Apple stellt selbst einen Compiler mit Entwicklungsumgebung zur Verfügung:

- Xcode[20], eine komplette Entwicklungsumgebung für: C, C++, Java und andere, die Mac OS X beiliegt.
- Apple's Programmer's Workshop[21] kostenlose Entwicklungsumgebung für MacOS 7 bis einschließlich 9.2.2.
- Gnu Compiler Collection[22] Gcc, wird über das Terminal gesteuert. Ist nach der Installation von Xcode [23] dabei.

10 http://www.mingw.org/
11 http://www.digitalmars.com/
12 http://cc.codegear.com/Free.aspx?id=24778
13 http://dn.codegear.com/article/21205
14 http://dn.codegear.com/article/20997
15 http://gcc.gnu.org
16 http://clang.llvm.org/
17 http://www.tinycc.org/
18 http://pcc.ludd.ltu.se/
19 http://software.intel.com/en-us/articles/intel-compilers/
20 http://developer.apple.com/tools/xcode/
21 http://developer.apple.com/tools/mpw-tools
22 http://gcc.gnu.org/
23 http://developer.apple.com/tools/xcode/

22.1.4 Amiga

- SAS/C, kommerziell
- vbcc, frei
- GCC

22.1.5 Atari

- GNU C Compiler[24], existiert auch in gepflegter Fassung für das freie Posix Betriebssystem MiNT, auch als Crosscompiler.
- AHCC[25], ein Pure-C kompatibler Compiler/Assembler, funktioniert auch unter SingleTOS und ist ebenfalls Open Source.

Neben diesen gibt es noch zahllose andere C-Compiler, von optimierten Intel- oder AMD-Compilern bis hin zu Compilern für ganz exotische Plattformen (cc65 für 6502).

22.2 GNU C Compiler

Der GNU C Compiler, Teil der GCC (GNU Compiler Collection), ist wohl der populärste Open-Source-C-Compiler und ist für viele verschiedene Plattformen verfügbar. Er ist in der GNU Compiler Collection enthalten und der Standard-Compiler für GNU/Linux und die BSD-Varianten.

Compileraufruf: *gcc Quellcode.c -o Programm*

Der GCC kompiliert und linkt nun die "Quellcode.c" und gibt es als "Programm" aus. Das Flag -c sorgt dafür, dass nicht gelinkt wird und bei -S wird auch nicht assembliert. Der GCC enthält nämlich einen eigenen Assembler, den GNU Assembler, der als Backend für die verschiedenen Compiler dient. Um Informationen über weitere Parameter zu erhalten, verwenden Sie bitte `man gcc`[26].

22.3 Microsoft Visual Studio

Die Microsoft Entwicklungsumgebung enthält eine eigene Dokumentation und ruft den Compiler nicht über die Kommandozeile auf, sondern ermöglicht die Bedienung über ihre Oberfläche.

Bevor Sie allerdings mit der Programmierung beginnen können, müssen Sie ein neues Projekt anlegen. Dazu wählen Sie in den Menüleiste den Eintrag "Datei" und "Neu..." aus. Im folgenden Fenster wählen Sie im Register "Projekte" den Eintrag "Win32-Konsolenanwendung" aus und geben einen Projektnamen ein. Verwechseln Sie nicht den Projektnamen mit dem

24 `http://vincent.riviere.free.fr/soft/m68k-atari-mint/`
25 `http://members.chello.nl/h.robbers/`
26 `http://www.manpage.org/cgi-bin/man/man2html?gcc-4.0+1`

Dateinamen! Die Endung .c darf hier deshalb noch nicht angegeben werden. Anschließen klicken Sie auf "OK" und "Fertigstellen" und nochmals auf "OK".

Nachdem Sie das Projekt erstellt haben, müssen Sie diesem noch eine Datei hinzufügen. Rufen Sie dazu nochmals den Menüeintrag "Datei" - "Neu..." auf und wählen Sie in der Registerkarte "Dateien" den Eintrag "C++ Quellcodedateien" aus. Dann geben Sie den Dateinamen ein, diesmal mit der Endung .c und bestätigen mit "OK". Der Dateiname muss nicht gleich dem Projektname sein.

In Visual Studio 6 ist das Kompilieren im Menü "Erstellen" unter "Alles neu erstellen" möglich. Das Programm können Sie anschließend in der "Eingabeaufforderung" von Windows ausführen.

en:C Programming/Compiling[27] es:Programación en C/Compilar un programa[28] et:Programmeerimiskeel C/Kompileerimine[29] fr:Programmation C-C++/Modularité et compilation[30] it:C/Compilatore e precompilatore/Compilatore[31] pt:Programar em C/Utilizando um compilador[32]

Die folgenden Zeichen sind in C erlaubt:

- Großbuchstaben:

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

- 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

- Ziffern:

0 1 2 3 4 5 6 7 8 9

- Sonderzeichen:

! " # % & ' ( ) * + , - . / : ; < = > ? [\ ] ^ _ { | } ~ *Leerzeichen*

- Steuerzeichen:

horizontaler Tabulator, vertikaler Tabulator, Form Feed

22.4 Ersetzungen

Der ANSI-Standard enthält außerdem so genannte Drei-Zeichen-Folgen (trigraph sequences), die der Präprozessor jeweils durch das im Folgenden angegebene Zeichen ersetzt. Diese Ersetzung erfolgt vor jeder anderen Bearbeitung.

27 http://en.wikibooks.org/wiki/C%20Programming%2FCompiling
28 http://es.wikibooks.org/wiki/Programaci%C3%B3n%20en%20C%2FCompilar%20un%20programa
29 http://et.wikibooks.org/wiki/Programmeerimiskeel%20C%2FKompileerimine
30 http://fr.wikibooks.org/wiki/Programmation%20C-C%2B%2B%2FModularit%C3%A9%20et%20compilation
31 http://it.wikibooks.org/wiki/C%2FCompilatore%20e%20precompilatore%2FCompilatore
32 http://pt.wikibooks.org/wiki/Programar%20em%20C%2FUtilizando%20um%20compilador

Drei-Zeichen-Folge =	**Ersetzung**
??=	#
??'	^
??-	~
??!	|
??/	\
??(	[
??)	]
??<	{
??>	}

ANSI C (C89)/ISO C (C90) Schlüsselwörter:

* auto* break[33]* case[34]* char* const[35]* continue[36]* default[37]* do[38]

* double* else[39]* enum[40]* extern* float* for[41]* goto[42]* if[43]

* int* long* register[44]* return[45]* short* signed* sizeof[46]* static[47]

* struct[48]* switch[49]* typedef* union[50]* unsigned* void* volatile[51]* while[52]

ISO C (C99) Schlüsselwörter:

* **_Bool*** **_Complex** * **_Imaginary*** **inline** * **restrict**

22.5 Ausdrücke

Ein Ausdruck ist eine Kombination aus Variablen, Konstanten, Operatoren und Rückgabewerten von Funktionen. Die Auswertung eines Ausdrucks ergibt einen Wert.

33 Kapitel 22.13.6 auf Seite 210
34 Kapitel 22.13.4 auf Seite 201
35 Kapitel 2.8 auf Seite 19
36 Kapitel 22.13.6 auf Seite 208
37 Kapitel 22.13.4 auf Seite 201
38 Kapitel 22.13.5 auf Seite 204
39 Kapitel 22.13.4 auf Seite 199
40 Kapitel 12.3 auf Seite 121
41 Kapitel 22.13.5 auf Seite 205
42 Kapitel 22.13.6 auf Seite 207
43 Kapitel 22.13.4 auf Seite 199
44 http://de.wikibooks.org/wiki/C-Programmierung%3A_static_%2526_Co.%23register
45 Kapitel 22.13.6 auf Seite 210
46 Kapitel 22.6.9 auf Seite 170
47 http://de.wikibooks.org/wiki/C-Programmierung%3A_static_%2526_Co.%23static
48 Kapitel 12.1 auf Seite 117
49 Kapitel 22.13.4 auf Seite 201
50 Kapitel 12.2 auf Seite 119
51 http://de.wikibooks.org/wiki/C-Programmierung%3A_static_%2526_Co.%23volatile
52 Kapitel 22.13.5 auf Seite 204

22.6 Operatoren

Man unterscheidet zwischen unären, binären und ternären Operatoren. Unäre Operatoren besitzen einen, binäre Operatoren besitzen zwei, ternäre drei Operanden. Die Operatoren *, &, + und – kommen sowohl als unäre wie auch als binäre Operatoren vor.

22.6.1 Vorzeichenoperatoren

Negatives Vorzeichen -

Liefert den negativen Wert eines Operanden. Der Operand muss ein arithmetischer Typ sein. Beispiel:

```
printf("-3 minus -2 = %i", -3 - -2); // Ergebnis ist -1
```

Positives Vorzeichen +

Der unäre Vorzeichenoperator `+`wurde in die Sprachdefinition aufgenommen, damit ein symmetrischer Operator zu `-`existiert. Er hat keine Einfluss auf den Operanden. So ist beispielsweise `+4.35`äquivalent zu `4.35`. Der Operand muss ein arithmetischer Typ sein. Beispiel:

```
printf("+3 plus +2= %i", +3 + +2); // Ergebnis ist 5
```

22.6.2 Arithmetik

Alle arithmetischen Operatoren, außer dem Modulo-Operator, können sowohl auf Ganzzahlen als auch auf Gleitkommazahlen angewandt werden. Arithmetische Operatoren sind immer binär.

Beim `+`und `-`Operator kann ein Operand auch ein Zeiger sein, der auf ein Objekt (etwa ein Array) verweist und der zweite Operand ein Integer sein. Das Resultat ist dann vom Typ des Zeigeroperanden. Wenn P auf das i-te Element eines Arrays zeigt, dann zeigt P + n auf das i+n-te Element des Array und P - n zeigt auf das i-n-te Element. Beispielsweise zeigt P + 1 auf das nächste Element des Arrays. Ist P bereits das letzte Element des Arrays, so verweist der Zeiger auf das nächste Element nach dem Array. Ist das Ergebnis nicht mehr ein Element des Arrays oder das erste Element nach dem Array, ist das Resultat undefiniert.

Addition +

Der Additionsoperator liefert die Summe der Operanden zurück. Beispiel:

```
int a = 3, b = 5;
int ergebnis;
ergebnis = a + b; // ergebnis hat den Wert 8
```

Subtraktion -

Der Subtraktionsoperator liefert die Differenz der Operanden zurück. Beispiel:

```
int a = 7, b = 2;
int ergebnis;
ergebnis = a - b; // ergebnis hat den Wert 5
```

Wenn zwei Zeiger subtrahiert werden, müssen beide Operanden Elemente desselben Arrays sein. Das Ergebnis ist vom Typ `ptrdiff`. Der Typ `ptrdiff`ist ein vorzeichenbehafteter Integerwert, der in der Headerdatei `<stddef.h>`definiert ist.

Multiplikation *

Der Multiplikationsoperator liefert das Produkt der beiden Operanden zurück. Beispiel:

```
int a = 5, b = 3;
int ergebnis;
ergebnis = a * b; // variable 'ergebnis' speichert den Wert 15
```

Division /

Der Divisionsoperator liefert den Quotienten aus der Division des ersten durch den zweiten Operanden zurück. Beispiel:

```
int a = 8, b = 2;
int ergebnis;
ergebnis = a/b; // Ergebnis hat den Wert 4
```

Bei einer Division durch 0 ist das Resultat undefiniert.

Aufgepasst bei Gleitkommazahlen
Funktionen wie 'printf' erwarten bei Berechnungen als Argumente ganze Zahlen. Darum führt eine einfache Division wie 3/5 als Argument zu einem falschen Resultat. Der Funktion müssen zwingend Gleitkommazahlen übergeben werden:

```
printf("3/5 = %f", 3.0/5.0 ); //Ergebnis ist 0.600000
```

Modulo %

Der Modulo-Operator liefert den Divisionsrest. Die Operanden des Modulo-Operators müssen vom ganzzahligen Typ sein. Beispiel:

```
int a = 5, b = 2;
int ergebnis;
ergebnis = a % b; // Ergebnis hat den Wert 1
```

Ist der zweite Operand eine 0, so ist das Resultat undefiniert.

22.6.3 Zuweisung

Der linke Operand einer Zuweisung muss ein modifizierbarer L-Wert sein.

Zuweisung =

Bei der einfachen Zuweisung erhält der linke Operand den Wert des rechten. Beispiel:

```
int a = 2, b = 3;
a = b; //a erhaelt Wert 3
```

Kombinierte Zuweisungen

Kombinierte Zuweisungen setzen sich aus einer Zuweisung und einer anderen Operation zusammen. Der Operand

```
a += b
```

wird zu

```
a = a + b
```

erweitert. Es existieren folgende kombinierte Zuweisungen:

`+=`, `-=`, `*=`, `/=`, `%=`, `&=`, `|=`, `^=`, `<<=`, `>>=`

Inkrement ++

Der Inkrement-Operator erhöht den Wert einer Variablen um 1. Wird er auf einen Zeiger angewendet, erhöht er dessen Wert um die Größe des Objekts, auf das der Zeiger verweist. Man unterscheidet Postfix (a++)- und Präfix (++a)-Notation. Bei der Postfix-Notation wird die Variable inkrementiert, nachdem sie verwendet wurde. Bei der Präfix-Notation wird sie inkrementiert, bevor sie verwendet wird. Die Notationsarten unterscheiden sich durch ihre Priorität (siehe Liste der Operatoren, geordnet nach ihrer Priorität[53]). Der Operand muss ein L-Wert sein.

53 Kapitel 22.6.10 auf Seite 172

Dekrement --

Der Dekrement-Operator verringert den Wert einer Variablen um 1. Wird er einen auf Zeiger angewendet, verringert er dessen Wert um die Größe des Objekts, auf das der Zeiger verweist. Auch hier unterscheidet man Postfix- und Präfix-Notation.

22.6.4 Vergleiche

Das Ergebnis eines Vergleichs ist 1, wenn der Vergleich zutrifft, andernfalls 0. Als Rückgabewert liefert der Vergleich einen integer-Wert. In C wird der boolsche Wert *true*durch einen Wert ungleich 0 und *false*durch 0 repräsentiert. Beispiel:

```
a = (4 == 3); // a erhaelt den Wert 0
a = (3 == 3); // a erhaelt den Wert 1
```

Gleichheit ==

Der Gleichheits-Operator vergleicht die beiden Operanden auf Gleichheit. Er besitzt einen geringeren Vorrang[54] als <, >, <= und >=.

Ungleichheit !=

Der Ungleichheits-Operator vergleicht die beiden Operanden auf Ungleichheit. Er besitzt einen geringeren Vorrang[55] als <, >, <= und >=.

Kleiner <

Der Kleiner als-Operator liefert dann 1, wenn der Wert des linken Operanden kleiner ist als der des Rechten. Beispiel:

```
int a = 7, b = 2;
int ergebnis;
ergebnis = a < b; // Ergebnis hat den Wert 0
ergebnis = b < a; // Ergebnis hat den Wert 1
```

Größer >

Der Größer als-Operator liefert dann 1, wenn der Wert des linken Operanden größer ist als der des Rechten. Beispiel:

```
int a = 7, b = 2;
int ergebnis;
ergebnis = a > b; // Ergebnis hat den Wert 1
ergebnis = b > a; // Ergebnis hat den Wert 0
```

54 Kapitel 22.6.10 auf Seite 172
55 Kapitel 22.6.10 auf Seite 172

Kleiner gleich <=

Der Kleiner gleich-Operator liefert dann 1, wenn der Wert des linken Operanden kleiner oder exakt gleich ist wie der Wert des Rechten. Beispiel:

```
int a = 2, b = 7, c = 7;
int ergebnis;
ergebnis = a <= b; // Ergebnis hat den Wert 1
ergebnis = b <= c; // Ergebnis hat ebenfalls den Wert 1
```

Größer gleich >=

Der Größer Gleich - Operator liefert dann 1, wenn der Wert des linken Operanden größer oder exakt gleich ist wie der Wert des Rechten. Beispiel:

```
int a = 2, b = 7, c = 7;
int ergebnis;
ergebnis = b >= a; // Ergebnis hat den Wert 1
ergebnis = b >= c; // Ergebnis hat ebenfalls den Wert 1
```

22.6.5 Aussagenlogik

Logisches NICHT !

Ist ein unärer Operator und invertiert den Wahrheitswert eines Operanden. Beispiel:

```
printf("Das logische NICHT liefert den Wert %i, wenn die Bedingung (nicht)
 erfuellt ist.", !(2<1)); //Ergebnis hat den Wert 1
```

Logisches UND &&

Das Ergebnis des Ausdrucks ist 1, wenn beide Operanden ungleich 0 sind, andernfalls 0. Im Unterschied zum `&`wird der Ausdruck streng von links nach rechts ausgewertet. Wenn der erste Operand bereits 0 ergibt, wird der zweite Operand nicht mehr ausgewertet und der Ausdruck liefert in jedem Fall den Wert 0. Nur wenn der erste Operand 1 ergibt, wird der zweite Operand ausgewertet. Der `&&`Operator ist ein Sequenzpunkt: Alle Nebenwirkungen des linken Operanden müssen bewertet worden sein, bevor die Nebenwirkungen des rechten Operanden ausgewertet werden.

Das Resultat des Ausdrucks ist vom Typ `int`. Beispiel:

```
printf("Das logische UND liefert den Wert %i, wenn beide Bedingungen erfuellt
 sind.", 2 > 1 && 3 < 4); //Ergebnis hat den Wert 1
```

Logisches ODER ||

Das Ergebnis ist 1, wenn einer der Operanden ungleich 0 ist, andernfalls ist es 0. Der Ausdruck wird streng von links nach rechts ausgewertet. Wenn der erste Operand einen von 0 verschieden Wert liefert, ist das Ergebnis des Ausdruck 1, und der zweite Operand wird nicht mehr ausgewertet. Auch dieser Operator ist ein Sequenzpunkt.

Das Resultat des Ausdrucks ist vom Typ `int`. Beispiel:

```
printf("Das logische ODER liefert den Wert %i, wenn eine der beiden Bedingungen
 erfuellt ist.", 2 > 3 || 3 < 4); // Ergebnis hat den Wert 1
```

22.6.6 Bitmanipulation

Bitweises UND / AND &

Mit dem UND-Operator werden zwei Operanden bitweise verknüpft. Die Verknüpfung darf nur für Integer-Operanden verwendet werden.

Wahrheitstabelle der UND-Verknüpfung:

b	**a**	**a & b**
falsch	falsch	falsch
falsch	wahr	falsch
wahr	falsch	falsch
wahr	wahr	wahr

Beispiel:

```
a = 45 & 35               // a == 33
```

Bitweises ODER / OR |

Mit dem ODER-Operator werden zwei Operanden bitweise verknüpft. Die Verknüpfung darf nur für Integer-Operanden verwendet werden.

Wahrheitstabelle der ODER-Verknüpfung:

a	**b**	**a | b**
falsch	falsch	falsch
falsch	wahr	wahr
wahr	falsch	wahr
wahr	wahr	wahr

Beispiel:

```
a = 45 | 35                 // a == 47
```

Bitweises exklusives ODER (XOR) ^

Mit dem XOR-Operator werden zwei Operanden bitweise verknüpft. Die Verknüpfung darf nur für Integer-Operanden verwendet werden.

Wahrheitstabelle der XOR-Verknüpfung:

a	**b**	**a ^ b**
falsch	falsch	falsch
falsch	wahr	wahr
wahr	falsch	wahr
wahr	wahr	falsch

Beispiel:

```
a = 45 ^ 35;                // a == 14
```

Bitweises NICHT / NOT ~

Mit der NICHT-Operation wird der Wahrheitswert eines Operanden bitweise umgekehrt

Wahrheitstabelle der NOT-Verknüpfung:

a	**~a**
101110	010001
111111	000000

Beispiel:

```
a = ~45
```

Linksshift <<

Verschiebt den Inhalt einer Variable bitweise nach links. Bei einer ganzen nicht negativen Zahl entspricht eine Verschiebung einer Multiplikation mit 2^n, wobei n die Anzahl der Verschiebungen ist, wenn das höchstwertige Bit nicht links hinausgeschoben wird. Das Ergebnis ist undefiniert, wenn der zu verschiebende Wert negativ ist.

Beispiel:

```
y = x << 1;
```

x	y
01010111	10101110

Rechtsshift >>

Verschiebt den Inhalt einer Variable bitweise nach rechts. Bei einer ganzen, nicht negativen Zahl entspricht eine Verschiebung einer Division durch 2^n und dem Abschneiden der Nachkommastellen (falls vorhanden), wobei n die Anzahl der Verschiebungen ist. Das Ergebnis ist implementierungsabhängig, wenn der zu verschiebende Wert negativ ist.

Beispiel:

```
y = x >> 1;
```

x	y
01010111	00101011

22.6.7 Datenzugriff

Dereferenzierung *

Der Dereferenzierungs-Operator (auch Indirektions-Operator oder Inhalts-Operator genannt) dient zum Zugriff auf ein Objekt durch einen Zeiger[56]. Beispiel:

```
int a;
int *zeiger;
zeiger = &a;
*zeiger = 3; // Setzt den Wert von a auf 3
```

Der unäre Dereferenzierungs-Operator bezieht sich immer auf den rechts stehenden Operanden.

Jeder Zeiger hat einen festgelegten Datentyp. Die Notation

```
int *zeiger
```

mit Leerzeichen zwischen dem Datentyp und dem Inhalts-Operator soll dies zum Ausdruck bringen. Eine Ausnahme bildet nur ein Zeiger vom Typ *void*. Ein so definierter Zeiger kann einen Zeiger beliebigen Typs aufnehmen. Zum schreiben muss der Datentyp per Typumwandlung[57] festgelegt werden.

Elementzugriff ->

Dieser Operator stellt eine Vereinfachung dar, um über einen Zeiger auf ein Element einer Struktur oder Union zuzugreifen.

```
objZeiger->element
```

entspricht

```
(*objZeiger).element
```

56 Kapitel 9 auf Seite 77
57 Kapitel 22.6.3 auf Seite 164

Elementzugriff .

Der Punkt-Operator dient dazu, auf Elemente einer Struktur oder Union zuzugreifen

22.6.8 Typumwandlung

Typumwandlung ()

Mit dem Typumwandlungs-Operator kann der Typ des Wertes einer Variable für die Weiterverarbeitung geändert werden, nicht jedoch der Typ einer Variable. Beispiel:

```
float f = 1.5;
int i = (int)f; // i erhaelt den Wert 1

float a = 5;
int b = 2;
float ergebnis;
ergebnis = a / (float)b; //ergebnis erhaelt den Wert 2.5
```

22.6.9 Speicherberechnung

Adresse &

Mit dem Adress-Operator erhält man die Adresse einer Variablen im Speicher. Das wird vor allem verwendet, um Zeiger[58] auf bestimmte Variablen verweisen zu lassen. Beispiel:

```
int *zeiger;
int a;
zeiger = &a; // zeiger verweist auf die Variable a
```

Der Operand muss ein L-Wert sein.

Speichergröße sizeof

Mit dem `sizeof`-Operator kann die Größe eines Datentyps oder eines Datenobjekts in Byte ermittelt werden. `sizeof`liefert einen ganzzahligen Wert ohne Vorzeichen zurück, dessen Typ `size_t`in der Headerdatei `stddef.h`[59] festgelegt ist.

Beispiel:

```
int a;
int groesse = sizeof(a);
```

Alternativ kann man sizeof als Parameter auch den Namen eines Datentyps übergeben. Dann würde die letzte Zeile wie folgt aussehen:

58 Kapitel 9 auf Seite 77
59 Kapitel 22.10.2 auf Seite 177

```
int groesse = sizeof(int);
```

Der Operator `sizeof`liefert die größe in Bytes zurück. Die Größe eines `int`beträgt mindestens 8 Bit, kann je nach Implementierung aber auch größer sein. Die tatsächliche Größe kann über das Macro `CHAR_BIT`, das in der Standardbibliothek `limits.h`[60]definiert ist, ermittelt werden. Der Ausdruck `sizeof(char)`liefert immer den Wert 1.

Wird `sizeof`auf ein Array angewendet, ist das Resultat die Größe des Arrays, `sizeof`auf ein Element eines Arrays angewendet liefert die Größe des Elements. Beispiel:

```
char a[10];
sizeof(a);    // liefert 10
sizeof(a[3]); // liefert 1
```

Der `sizeof`-Operator darf nicht auf Funktionen oder Bitfelder angewendet werden.

22.6.10 Sonstige

Funktionsaufruf ()

Bei einem Funktionsaufruf stehen nach dem Namen der Funktion zwei runde Klammern. Wenn Parameter übergeben werden, stehen diese zwischen diesen Klammern. Beispiel:

```
funktion(); // Ruft funktion ohne Parameter auf
funktion2(4, a); // Ruft funktion2 mit 4 als ersten und a als zweiten Parameter
 auf
```

Komma-Operator ,

Der Komma-Operator erlaubt es, zwei Ausdrücke auszuführen, wo nur einer erlaubt wäre. Die Ergebnisse aller durch diesen Operator verknüpften Ausdrücke außer dem letzten werden verworfen. Am häufigsten wird er in For-Schleifen verwendet, wenn zwei Schleifen-Variablen vorhanden sind.

```
int x = (1,2,3); // entspricht  int x = 3;
for (i=0,j=1; i<10; i++, j--)
{
   //...
}
```

Bedingung ?:

Der Bedingungs-Operator, auch als ternärer Operator bezeichnet, hat drei Operanden und folgende Syntax

60 Kapitel 22.10.2 auf Seite 177

```
Bedingung ? Ausdruck1 : Ausdruck2
```

Zuerst wird die Bedingung ausgewertet. Trifft diese zu, wird der erste Ausdruck abgearbeitet, andernfalls der zweite. Beispiel:

```
int a, b, max;
a = 5;
b = 3;
max = (a > b) ? a : b; //max erhalt den Wert von a (also 5),
                       //weil diese die Variable mit dem größeren Wert ist
```

Indizierung []

Der Index-Operator wird verwendet, um ein Element eines Arrays[61] anzusprechen. Beispiel:

```
a[3] = 5;
```

Klammerung ()

Geklammerte Ausdrücke werden vor den anderen ausgewertet. Dabei folgt C den Regeln der Mathematik, dass innere Klammern zuerst ausgewertet werden. So durchbricht

```
ergebnis = (a + b) * c
```

die Punkt-vor-Strich-Regel, die sonst bei

```
ergebnis = a + b * c
```

gelten würde.

Liste der Operatoren, geordnet nach absteigender Priorität sowie deren Assoziativität

Priorität	Symbol	Assoziativität[62]	Bedeutung
15	(Postfix) ++	L - R	Postfix-Inkrement[63]
	(Postfix) --		Postfix-Dekrement[64]
	()		Funktionsaufruf[65]
	[]		Indizierung[66]
	->		Elementzugriff[67]
	.		Elementzugriff[68]

61 Kapitel 10 auf Seite 85
63 Kapitel 22.6.3 auf Seite 164
64 Kapitel 22.6.3 auf Seite 165
65 Kapitel 22.6.3 auf Seite 164
66 Kapitel 22.6.3 auf Seite 164
67 Kapitel 22.6.3 auf Seite 164
68 Kapitel 22.6.7 auf Seite 170

Priorität	Symbol	Assoziativität[62]	Bedeutung
	(Typ){Initialisierungsliste}		compound literal (C99)
14	++ (Präfix)	R - L	Präfix-Inkrement[69]
	--(Präfix)		Präfix-Dekrement[70]
	+ (Vorzeichen)		Vorzeichen[71]
	-(Vorzeichen)		Vorzeichen[72]
	!		logisches NICHT[73]
	~		bitweises NICHT[74]
	&		Adresse[75]
	*		Zeigerdereferenzierung[76]
	(Typ)		Typumwandlung[77]
	sizeof		Speichergröße[78]
	_Alignof		alignment requirement (C11)
13	*	L - R	Multiplikation[79]
	/		Division[80]
	%		Modulo[81]
12	+	L - R	Addition[82]
	-		Subtraktion[83]
11	<<	L - R	Links-Shift[84]
	>>		Rechtsshift[85]
10	<	L - R	kleiner[86]
	<=		kleiner gleich[87]
	>		größer[88]

69 Kapitel 22.6.3 auf Seite 164
70 Kapitel 22.6.3 auf Seite 165
71 Kapitel 22.6.3 auf Seite 164
72 Kapitel 22.6.1 auf Seite 162
73 Kapitel 22.6.3 auf Seite 164
74 http://de.wikibooks.org/wiki/C-Programmierung%3A_Ausdr%25C3%25BCcke_und_Operatoren%23Bitweises_NICHT_.2F_NOT_.7E
75 http://de.wikibooks.org/wiki/C-Programmierung%3A_Ausdr%25C3%25BCcke_und_Operatoren%23Adresse_.26
76 Kapitel 22.6.3 auf Seite 164
77 Kapitel 22.6.3 auf Seite 164
78 http://de.wikibooks.org/wiki/C-Programmierung%3A_Ausdr%25C3%25BCcke_und_Operatoren%23Speichergr.C3.B6.C3.9Fe_sizeof
79 Kapitel 22.6.3 auf Seite 164
80 Kapitel 22.6.3 auf Seite 164
81 Kapitel 22.6.3 auf Seite 164
82 Kapitel 22.6.3 auf Seite 164
83 Kapitel 22.6.2 auf Seite 163
84 Kapitel 22.6.3 auf Seite 164
85 Kapitel 22.6.3 auf Seite 164
86 Kapitel 22.6.3 auf Seite 164
87 Kapitel 22.6.3 auf Seite 164
88 Kapitel 22.6.3 auf Seite 164

Priorität	Symbol	Assoziativität[62]	Bedeutung
	>=		größer gleich[89]
9	==	L - R	gleich[90]
	!=		ungleich[91]
8	&	L - R	bitweises UND[92]
7	^	L - R	bitweises exklusives ODER[93]
6	|	L - R	bitweises ODER[94]
5	&&	L - R	logisches UND[95]
4	||	L - R	logisches ODER[96]
3	?:	R - L	Bedingung[97]
2	=	R - L	Zuweisung[98]
	*=, /=, %=, +=, -=, &=, ^=, |=, <<=, >>=		Zusammengesetzte Zuweisung[99]
1	,	L - R	Komma-Operator[100]

89 Kapitel 22.6.3 auf Seite 164
90 Kapitel 22.6.3 auf Seite 164
91 Kapitel 22.6.3 auf Seite 164
92 http://de.wikibooks.org/wiki/C-Programmierung%3A_Ausdr%25C3%25BCcke_und_Operatoren%23Bitweises_UND_.2F_AND_.26
93 http://de.wikibooks.org/wiki/C-Programmierung%3A_Ausdr%25C3%25BCcke_und_Operatoren%23Bitweises_exklusives_ODER_.28XOR.29_.5E
94 http://de.wikibooks.org/wiki/C-Programmierung%3A_Ausdr%25C3%25BCcke_und_Operatoren%23Bitweises_ODER_.2F_OR_.7C
95 Kapitel 22.6.3 auf Seite 164
96 Kapitel 22.6.3 auf Seite 164
97 Kapitel 22.6.3 auf Seite 164
98 Kapitel 22.6.3 auf Seite 164
99 Kapitel 22.6.3 auf Seite 164
100 Kapitel 22.6.3 auf Seite 164

22.7 Grunddatentypen

22.7.1 Ganzzahlen

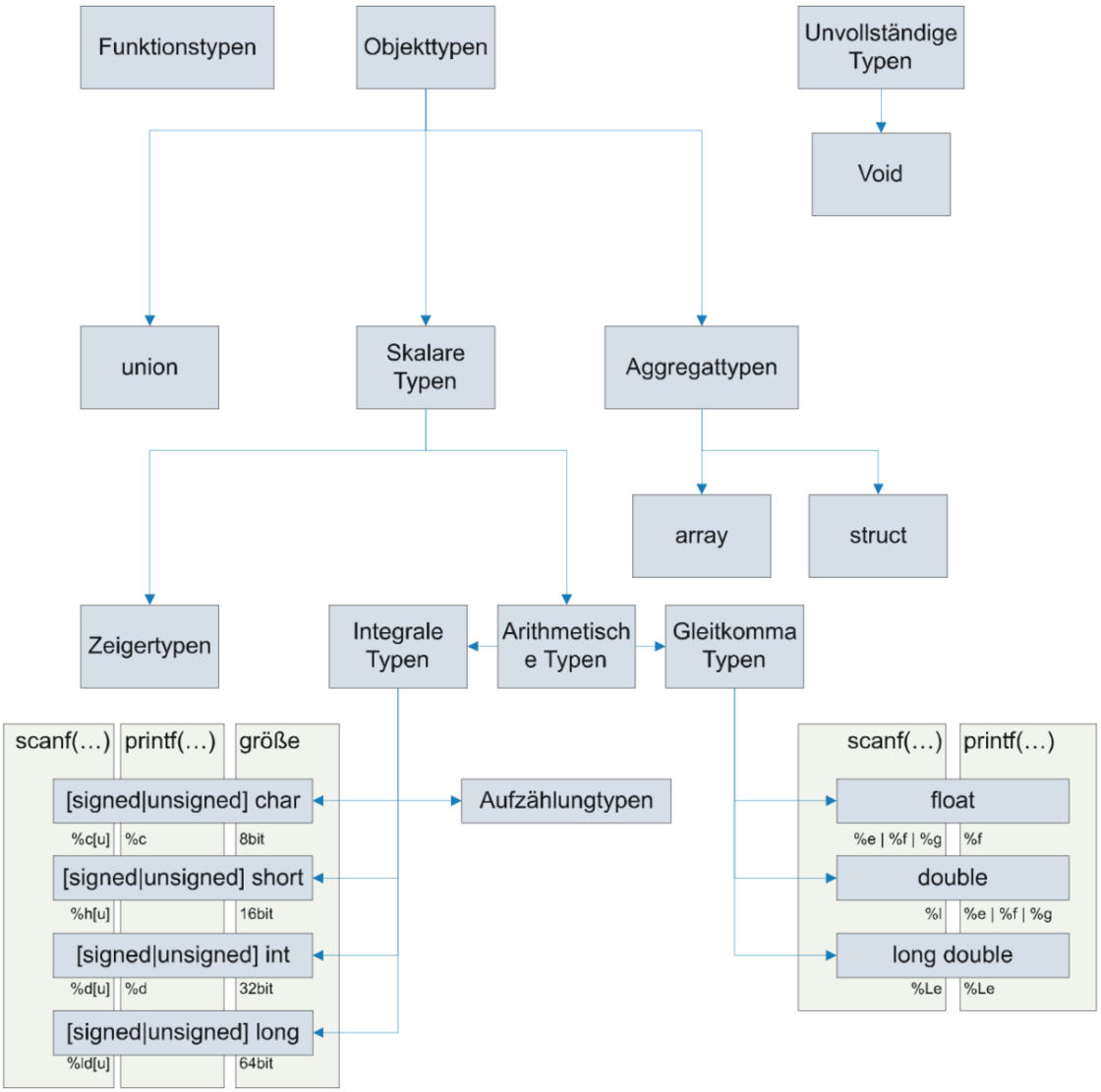


Abb. 7 Grafische Darstellung der Datentypen in C

Typ	**Vorzeichenbehaftet**	**Vorzeichenlos**
char	-128 bis 127	0 bis 255
short int	-32.768 bis 32.767	0 bis 65.535
long int	-2.147.483.648 bis 2.147.483.647	0 bis 4.294.967.295
long long int	-9.223.372.036.854.775.808 bis 9.223.372.036.854.775.807	0 bis 18.446.744.073.709.551.615

Alle angegeben Werte sind Mindestgrößen. Die in der Implementierung tatsächlich verwendeten Größen sind in der Headerdatei limits.h[101] definiert.

Auf Maschinen, auf denen negative Zahlen im Zweierkomplement dargestellt werden, erhöht sich der negative Zahlenbereich um eins. Deshalb ist beispielsweise der Wertebereich für den Typ `signed char`bei den meisten Implementierungen zwischen -128 und +127.

Eine ganzzahlige Variable wird mit dem Schlüsselwort `unsigned`als vorzeichenlos vereinbart, mit dem Schlüsselwort `signed`als vorzeichenbehaftet. Fehlt diese Angabe, so ist die Variable vorzeichenbehaftet, beim Datentyp `char`ist dies implementierungsabhängig.

Der Typ `int`besitzt laut Standard eine "natürliche Größe". Allerdings muss `short`kleiner oder gleich groß wie `int`und `int`muss kleiner oder gleich groß wie `long`sein.

Der Standard legt fest, dass `char`groß genug sein muss, um alle Zeichen aus dem Standardzeichensatz aufnehmen zu können. Wird ein Zeichen gespeichert, so garantiert der Standard, dass `char`vorzeichenlos ist.

Mit dem C99-Standard wurde der Typ `_Bool`eingeführt. Er kann die Werte 0 (false) und 1 (true) aufnehmen. Wie groß der Typ ist, schreibt der ANSI-Standard nicht vor, allerdings muss `_Bool`groß genug sein, um 0 und 1 zu speichern. Wird ein Wert per "cast" in den Datentyp `_Bool`umgewandelt, dann ist das Ergebnis 0, wenn der umzuwandelnde Wert 0 ist, andernfalls ist das Ergebnis 1.

22.7.2 Fließkommazahlen

Die von der Implementierung verwendeten Zahlen sind in der Headerdatei <float.h>[102] definiert.

22.8 Größe eines Typs ermitteln

Die Größe eines Typs auf einem System wird mit dem `sizeof`-Operator ermittelt. Siehe Referenzkapitel Operatoren[103]. `sizeof` *`typ`*gibt aber nicht, wie oft vermutet, die Größe einer Variable dieses Typs in Bytes zurück, sondern nur, um welchen Faktor eine solche Variable größer als eine byte-Variable ist. Da jedoch byte auf den meisten Implementierungen ein Byte belegt, stimmen diese Werte meistens überein.

Inhaltsverzeichnis[104]

22.9 Einführung in die Standard Header

Die 16 ANSI C (C89) und 3 weiteren ISO C (C94/95) Header sind auch ein Teil der C++ Standard Template Library, die neuen ISO C (C99) jedoch nicht. Wer gezwungen ist einen

101 Kapitel 22.10.2 auf Seite 177
102 Kapitel 22.10.2 auf Seite 177
103 Kapitel 22.6.9 auf Seite 170
104 `http://de.wikibooks.org/wiki/C-Programmierung`

C++ Compiler zu benutzen oder daran denkt, sein Programm später von C nach C++ zu portieren, sollte die C99-Erweiterungen nicht benutzen.

Weitere Hintergrundinformationen zur Standardbibliothek finden Sie in der Wikipedia[105].

22.10 ANSI C (C89)/ISO C (C90) Header

22.10.1 assert.h

Testmöglichkeiten und Fehlersuche.

22.10.2 ctype.h

Die Datei ctype.h enthält diverse Funktionen mit denen sich einzelne Zeichen überprüfen lassen oder umgewandelt werden können.

Übersicht

Der Header ctype.h enthält diverse Funktionen, mit denen sich einzelne Zeichen überprüfen lassen oder umgewandelt werden können. Die Funktionen liefern einen von 0 verschiedenen Wert, wenn *c*die Bedingung erfüllt, andernfalls liefern sie 0:

- `int isalnum(int c)`testet auf alphanumerisches Zeichen (a-z, A-Z, 0-9)
- `int isalpha(int c)`testet auf Buchstabe (a-z, A-Z)
- `int iscntrl(int c)`testet auf Steuerzeichen (\f, \n, \t ...)
- `int isdigit(int c)`testet auf Dezimalziffer (0-9)
- `int isgraph(int c)`testet auf druckbare Zeichen ohne Leerzeichen
- `int islower(int c)`testet auf Kleinbuchstaben (a-z)
- `int isprint(int c)`testet auf druckbare Zeichen mit Leerzeichen
- `int ispunct(int c)`testet auf druckbare Interpunktionszeichen
- `int isspace(int c)`testet auf Zwischenraumzeichen (Leerzeichen, \f, \n, \t ...)
- `int isupper(int c)`testet auf Grossbuchstaben (A-Z)
- `int isxdigit(int c)`testet auf hexadezimale Ziffern (0-9, a-f, A-F)

Zusätzlich sind noch zwei Funktionen für die Umwandlung in Groß- bzw. Kleinbuchstaben definiert:

- `int tolower(int c)`wandelt Gross- in Kleinbuchstaben um
- `int toupper(int c)`wandelt Klein- in Grossbuchstaben um

Häufig gemachte Fehler

Wie Sie vielleicht sehen, erwarten die Funktionen aus <`ctype.h`>als Parameter einen *int*, obwohl es eigentlich ein *char*sein sollte. Immerhin arbeiten die Funktionen ja mit Zeichen.

105 `http://de.wikipedia.org/wiki/Standard%20C%20Library%20`

Die Ursache hierfür liegt im C-Standard selbst. Laut C-Standard muss *c*entweder »als *unsigned char*repräsentierbar oder der Wert des Makros *EOF*sein«. Ansonsten ist das Verhalten undefiniert. *EOF*ist im Standard als negativer int-Wert definiert, unsigned char kann aber niemals negative Werte annnehmen. Um dem Standard zu genügen, muss also ein ausreichender Parametertyp deklariert werden, der sowohl den unsigned char-Wertebereich wie auch negative int-Werte abbilden kann. Dies kann der Basisdatentyp int.

Das alleine ist noch nicht schlimm. Aber: in C gibt es *drei*verschiedene Arten von *char*-Datentypen: *char*, *signed char*und *unsigned char*. In einer Umgebung mit Zweierkomplementdarstellung, in der ein *char*8 Bit groß ist (ja, es gibt auch andere), geht der Wertebereich von *signed char*von -128 bis +127, der von *unsigned char*von 0 bis 255. Gerade auf der i386-Architektur ist es üblich, *char*mit *signed char*gleichzusetzen. Wenn man jetzt noch annimmt, dass der Zeichensatz ISO-8859-1 (latin1) oder Unicode/UTF-8 ist, darf man diesen Funktionen keine Strings übergeben, die möglicherweise Umlaute enthalten. Ein typisches Beispiel, bei dem das dennoch geschieht, ist:

```
int all_spaces(const char *s)
{
  while (*s != '\0') {
    if (isspace(*s))       /* FEHLER */
      return 0;
    s++;
  }
  return 1;
}
```

Der Aufruf von *all_ spaces("Hallöle")*führt dann zu undefiniertem Verhalten. Um das zu vermeiden, muss man das Argument der Funktion *isspace*in einen *unsigned char*umwandeln. Das geht zum Beispiel so:

```
    if (isspace((unsigned char) *s))
```

22.10.3 errno.h

Die Headerdatei enthält Funktionen zum Umgang mit Fehlermeldungen und die globale Variable errno, welche die Fehlernummer des zuletzt aufgetretenen Fehlers implementiert.

22.10.4 float.h

Die Datei float.h enthält Definitionen zur Bearbeitung von Fließkommazahlen in C.

Der Standard definiert eine Gleitkommazahl nach dem folgenden Modell (in Klammern die symbolischen Konstanten für den Typ float):

$x = sb^e \sum_{k=1}^{p} f_k \cdot b^{-k}, e_{min} \leq e \leq e_{max}$

- s = Vorzeichen
- b = Basis (`FLT_RADIX`)
- e = Exponent (Wert zwischen `FLT_MIN`und `FLT_MAX`)

- p = Genauigkeit (`FLT_MANT_DIG`)
- f_k = nichtnegative Ganzzahl kleiner b

Der Standard weist darauf hin, dass hierbei nur um ein Beschreibung der Implementierung von Fließkommazahlen handelt und sich von der tatsächlichen Implementierung unterscheidet.

Mit float.h stehen folgende **Gleitkommatypen**zur Verfügung:

- float
- double
- long double

Für alle Gleitkommatypen definierte symbolische Konstanten:

- `FLT_RADIX`(2) Basis
- `FLT_ROUND`Erhält die Art der Rundung einer Implementierung:
  - -1 unbestimmt
  - 0 in Richtung 0
  - 1 zum nächsten Wert
  - 2 in Richtung plus unendlich
  - 3 in Richtung minus unendlich

Die symbolische Konstante `FLT_ROUND`kann auch andere Werte annehmen, wenn die Implementierung ein anderes Rundungsverfahren benutzt.

Für den Typ Float sind definiert:

- `FLT_MANT_DIG`Anzahl der Ziffern in der Mantisse
- `FLT_DIG`(6) Genauigkeit in Dezimalziffern
- `FLT_EPSILON`(1E-5) kleinste Zahl x für die gilt 1.0 + x ≠1.0
- `FLT_MAX`(1E+37) größte Zahl, die der Typ float darstellen kann
- `FLT_MIN`(1E-37) kleinste Zahl größer als 0, die der Typ float noch darstellen kann
- `FLT_MAX_EXP`Minimale Größe des Exponent
- `FLT_MIN_EXP`Maximale Größe des Exponent

Für den Typ Double sind definiert:

- `DBL_MANT_DIG`Anzahl der Ziffern in der Matisse
- `DBL_DIG`(10) Genauigkeit in Dezimalziffern
- `DBL_EPSILON`(1E-9) kleinste Zahl x für die gilt 1.0 + x ≠1.0
- `DBL_MAX`(1E+37) größte Zahl, die der Typ double darstellen kann
- `DBL_MIN`(1E-37) kleinste Zahl größer als 0, die der Typ double noch darstellen kann
- `DBL_MAX_EXP`Minimale Größe des Exponent
- `DBL_MIN_EXP`Maximale Größe des Exponent

Für den Typ Long Double sind definiert:

- `LDBL_MANT_DIG`Anzahl der Ziffern in der Matisse
- `LDBL_DIG`(10) Genauigkeit in Dezimalziffern
- `LDBL_EPSILON`(1E-9) kleinste Zahl x für die gilt 1.0 + x ≠1.0
- `LDBL_MAX`(1E+37) größte Zahl, die der Typ long double darstellen kann
- `LDBL_MIN`(1E-37) kleinste Zahl größer als 0, die der Typ long double noch darstellen kann
- `LDBL_MAX_EXP`Minimale Größe des Exponent

- `LDBL_MIN_EXP`Maximale Größe des Exponent

22.10.5 limits.h

Enthält die implementierungsspezifischen Minimal- und Maximalwerte für die einzelnen Datentypen.

Die Headerdatei erhält die Werte, die ein Typ auf einer bestimmten Implementierung annehmen kann. In Klammern befinden sich die jeweiligen Mindestgrößen. Für den Typ `char`sind zwei unterschiedliche Größen angegeben, da es von der Implementierung abhängig ist, ob dieser vorzeichenbehaftet oder vorzeichenlos ist. Der Wertebereich ist immer asymmetrisch (z. B. -128, +127).

- `CHAR_BIT`Anzahl der Bits in einem **char**(8 Bit)
- `SCHAR_MIN`minimaler Wert, den der Typ **signed char**aufnehmen kann (-128)
- `SCHAR_MAX`maximaler Wert, den der Typ **signed char**aufnehmen kann (+127)
- `UCHAR_MAX`maximaler Wert, den der Typ **unsigned char**aufnehmen kann(+255)
- `CHAR_MIN`minimaler Wert, den die Variable **char**aufnehmen kann (0 oder SCHAR_MIN)
- `CHAR_MAX`maximaler Wert, den die Typ **char**aufnehmen kann (SCHAR_MAX oder UCHAR_MAX)
- `SHRT_MIN`minimaler Wert, den der Typ **short int**annehmen kann (-32.768)
- `SHRT_MAX`maximaler Wert, den der Typ **short int**annehmen kann (+32.767)
- `USHRT_MAX`maximaler Wert, den der Typ **unsigned short int**annehmen kann (+65.535)
- `INT_MIN`minimaler Wert, den der Typ **int**annehmen kann (-32.768)
- `INT_MAX`maximaler Wert, den der Typ **int**annehmen kann (+32.767)
- `UINT_MAX`maximaler Wert, den der Typ **unsigned int**aufnehmen kann(+65.535)
- `LONG_MIN`minimaler Wert, den der Typ **long int**annehmen kann (-2.147.483.648)
- `LONG_MAX`maximaler Wert, den der Typ **long int**annehmen kann (+2.147.483.647)
- `ULONG_MAX`maximaler Wert, den der Typ **unsigned long int**annehmen kann (+4.294.967.295)
- `LLONG_MIN`minimaler Wert, den der Typ **long long int**annehmen kann (–9.223.372.036.854.775.808)
- `LLONG_MAX`maximaler Wert, den der Typ **long long int**annehmen kann (+9.223.372.036.854.775.807)
- `ULLONG_MAX`maximaler Wert, den der Typ **unsigned long long int**annehmen kann (+18.446.744.073.709.551.615)

22.10.6 locale.h

Länderspezifische Eigenheiten wie Formatierungen und Geldbeträge.

22.10.7 math.h

Die Datei math.h enthält diverse höhere mathematische Funktionen, wie z.B. die Wurzeln, Potenzen, Logarithmen und anderes. Sie wird für Berechnungen gebraucht, welche nicht, oder nur umständlich, mit den Operatoren +, -, *, /, % ausgerechnet werden können.

Trigonometrische Funktionen:

- `double cos(double x)`Kosinus von x
- `double sin(double x)`Sinus von x
- `double tan(double x)`Tangens von x
- `double acos(double x)`arccos(x)
- `double asin(double x)`arcsin(x)
- `double atan(double x)`arctan(x)
- `double cosh(double x)`Cosinus Hyperbolicus von x
- `double sinh(double x)`Sinus Hyperbolicus von x
- `double tanh(double x)`Tangens Hyperbolicus von x

Logarithmusfunktionen:

- `double exp(double x)`Exponentialfunktion (e hoch x)
- `double log(double x)`natürlicher Logarithmus (Basis e)
- `double log10(double x)`dekadischer Logarithmus (Basis 10)

Potenzfunktionen:

- `double sqrt(double x)`Quadratwurzel von x
- `double pow(double x, double y)`Berechnet x^y

22.10.8 setjmp.h

Ermöglicht Funktionssprünge.

22.10.9 signal.h

Ermöglicht das Reagieren auf unvorhersehbare Ereignisse.

Die Datei signal.h enthält Makros für bestimmte Ereignisse, wie Laufzeitfehler und Unterbrechungsanforderungen und Funktionen, um auf diese Ereignisse zu reagieren.

22.10.10 stdarg.h

Die Datei stdarg.h enthält Makros und einen Datentyp zum Arbeiten mit variablen Parameterlisten.

- `va_list`

  Datentyp für einen Zeiger auf eine variable Parameterliste

- `void va_start(va_list par_liste, letzter_par);`

  initialisiert die Parameterliste anhand des letzten Parameters *letzter_ par*und assoziiert sie mit *par_ liste*

- `type va_arg(va_list par_liste, type);`

  liefert den nächsten Parameter der mit *par_ liste*assoziierten Parameterliste mit dem spezifiziertem Typ *type*zurück

- `void va_end(va_list par_liste);`

  gibt den von der variablen Parameterlist *par_ liste*belegten Speicherplatz frei

22.10.11 stddef.h

Allgemein benötigte Definitionen.

`size_t`

Implementierungsanhängiger, vorzeichenloser, ganzzahliger Variablentyp.

`NULL`

Das Macro repäsentiert ein Speicherbereich, der nicht gültig ist. Eine mögliche Implementierung des Macro lautet:

```
#ifndef NULL
 #define NULL ((void *) 0)
#endif
```

22.10.12 stdio.h

Die Datei stdio.h enthält Funktionen zum Arbeiten mit Dateien und zur formatierten und unformatierten Eingabe und Ausgabe von Zeichenketten.

Die Datei stdio.h enthält diverse Standard-Input-Output-Funktionen (daher der Name).

`FILE`

Vordefinierter Standardtyp zur Verwendung in vielen Funktionen zur Ein- und Ausgabe bei Streams bzw. Dateien. Meistens erfolgt die Instanziierung einer Variable für diesen Typ als Zeiger.

`NULL`

Das Macro für einen Pointer auf einen nicht existieren Speicherbereich wie es auch in stddef.h[106] definiert ist.

106 Kapitel 22.10.11 auf Seite 182

`BUFSIZE`

dieses Macro definiert die implementierungsspezifische Maximalgröße, die mit setbuf gewählt werden kann als Integerwert.

`FOPEN_MAX`

Enthält als Integer die Anzahl der möglichen gleichzeitig geöffneten Filepointer, welche diese Implementierung erlaubt.

`FILENAME_MAX`

Enthält als Integerwert die Maximallänge von Dateinamen mit dem die Implementierung sicher umgehen kann.

`stdin`

Ist ein immer vorhandener geöffneter Filepointer auf den Standardeingabe-Stream.

`stdout`

Ist ein immer vorhandener geöffneter Filepointer auf den Standardausgabe-Stream.

`stderr`

Ist ein immer vorhandener geöffneter Filepointer auf den Fehlerausgabe-Stream. Lassen Sie sich bitte nicht dadurch verwirren, dass meistens stderr auch auf der Konsole landet: Der Stream ist **nicht**der gleiche wie stdout.

`EOF`(End of File)

negativer Wert vom Typ `int`, der von einigen Funktionen zurückgegeben wird wenn das Ende eines Streams erreicht wurde.

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

entspricht `fprintf(stdout, const char* format, ...)`

Die Umwandlungsangaben (engl. conversion specification) bestehen aus den folgenden Elementen:

- dem `%`Zeichen
- Flags
- der Feldbreite (engl. field width), Anzahl der Druckstellen insgesamt

- der Genauigkeit (engl. precision), Zahl der Stellen nach dem Dezimalpunkt (bzw bei `%g`: Zahl der relevanten Stellen)
- einem Längenhinweis (engl. length modifier), Art der internen Zahlspeicherung (Sonderfall)
- dem Umwandlungszeichen (engl. conversion key)

Die Flags haben die folgende Bedeutung:

- `-`(Minus): Der Text wird linksbündig ausgerichtet. Wird das Flag nicht gesetzt so wird der Text rechtsbündig ausgerichtet.
- `+`(Plus): Es wird in jedem Fall ein Vorzeichen ausgegeben und zwar auch dann, wenn die Zahl positiv ist.
- Leerzeichen: Ein Leerzeichen wird ausgegeben. Wenn sowohl + wie auch das Leerzeichen benutzt werden, so wird die Kennzeichnung nicht beachtet und es wird kein Leerzeichen ausgegeben.
- `#`: Welche Wirkung das Kennzeichen # hat ist abhängig vom verwendeten Format: Wenn ein Wert über %x als Hexadezimal ausgegeben wird, so wird jedem Wert ein 0x vorangestellt (außer der Wert ist 0).
- `0`: Die Auffüllung erfolgt mit führenden Nullen anstelle von Leerzeichen (sofern Minus-Flag nicht gesetzt).

Mögliche Umwandlungszeichen:

- `%a`, `%A`: `double`im Format `[-]0xh.hhhhp±d`. Wird `%a`verwendet, so werden die Buchstaben a bis f als abcdef ausgegeben; wenn `%A`verwendet wird, dann werden die Buchstaben a bis f als ABCDEF ausgegeben (neu im C99 Standard).
- `%c`: `int`umgewandelt in `char`und als Zeichen interpretiert.
- `%d`, `%i`: `int`im Format `[-]dddd`.
- `%e`, `%E`: `double`in Format `[-]d.ddd e±dd`bzw. `[-]d.ddd E±dd`. Die Anzahl der Stellen nach dem Dezimalpunkt entspricht der Genauigkeit. Fehlt die Angabe, so ist sie standardmäßig 6. Ist die Genauigkeit 0 und das `#`Flag nicht gesetzt, so entfällt der Dezimalpunkt. Der Exponent besteht immer aus mindestens zwei Ziffern. Sind mehr Ziffern zur Darstellung notwendig, so werden nur so viele wie unbedingt notwendig angezeigt. Wenn der darzustellende Wert 0 ist, so ist auch der Exponent 0.
- `%f`, `%F`: `double`im Format `[-]ddd.ddd`. Die Anzahl der Stellen nach dem Dezimalpunkt entspricht der Genauigkeit. Fehlt die Angabe, so ist sie standardmäßig 6. Ist die Genauigkeit 0 und das `#`Flag nicht gesetzt, so entfällt der Dezimalpunkt.
- `%g`, `%G`: Ein `double`Argument wird im Stil von `e`bzw. `E`ausgegeben, allerdings nur, wenn der Exponent kleiner als -4 ist oder größer / gleich der Genauigkeit. Ansonsten wird das Argument im Stil von `%f`ausgegeben.
- `%n`: Das Argument muss ein vorzeichenbehafteter Zeiger sein, bei dem die Anzahl der auf dem Ausgabestrom geschriebenen Zeichen abgelegt wird.
- `%o`: `int`als Oktalzahl im Format `[-]dddd`.
- `%p`: `void*`. Der Wert des Zeigers umgewandelt in eine darstellbare Folge von Zeichen, wobei die genaue Darstellung von der Implementierung abhängig ist.
- `%s`: Das Argumente sollten ein Zeiger auf das erste Element eines Zeichenarray sein. Die nachfolgenden Zeichen werden bis zum \0 ausgegeben.
- `%u`: `unsigned int`im Format `dddd`
- `%X`, `%x`: `int`im Hexadezimalsystem im Format `[-]dddd`. Wird `%x`verwendet, so werden die Buchstaben a bis f als abcdef ausgegeben, wenn `%X`verwendet wird, dann werden die

Buchstaben a bis f als ABCDEF ausgegeben. Ist das #Flag gesetzt, dann erscheint die Ausgabe der Hexadezimalzahl mit einem vorangestellten "0x" (außer der Wert ist 0).
- %%Ein Prozentzeichen wird ausgegeben

Wenn eine Umwandlungsangabe ungültig ist oder ein Argumenttyp nicht dem Umwandlungsschlüssel entspricht, so ist das Verhalten undefiniert.

int fprintf(FILE *fp,const char *format,...)

Die Funktion macht das gleiche wie die Funktion `printf`, Ausgabe aber nicht nach stdout sondern in einen Stream, der über den Filepointer fp übergeben wird.

int snprintf(char *dest, size_t destsize, const char *format, ...)

Die Funktion `snprintf()`formatiert die in *...*angegebenen Argumente gemäß der *printf*-Formatierungsvorschrift *format*und schreibt das Ergebnis in den durch *dest*angegebenen String. *destsize*gibt die Maximallänge des Strings *dest*an. Der String in *dest*erhält in jedem Fall ein abschließendes Nullzeichen. In keinem Fall wird über *dest[destsize - 1]*hinausgeschrieben.

Der Rückgabewert ist die Anzahl der Zeichen, die geschrieben worden wäre, wenn der String *dest*lang genug gewesen wäre.

Um Pufferüberläufe zu vermeiden, sollte diese Funktion gegenüber *strcpy*, *strcat*, *strncpy*und *strncat*vorgezogen werden, da es bei letzteren Funktionen aufwendig ist, über den noch verfügbaren Platz im String Buch zu führen.

Beispielcode

Das folgende Programm erwartet zwei Argumente auf der Kommandozeile. Diese Argumente werden zu einem Dateinamen, bestehend aus Verzeichnisname und Dateiname, zusammengesetzt und ausgegeben. Falls der Dateiname zu lang für den Puffer wird, wird eine Fehlermeldung ausgegeben.

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

int main(int argc, char **argv)
{
        char fname[256];

        if (argc == 3) {
                if (snprintf(fname, sizeof(fname), "%s/%s", argv[1], argv[2]) >=
 sizeof(fname)) {
                        fprintf(stderr, "Fehler: Dateiname zu lang.\n");
                        exit(EXIT_FAILURE);
                }
                printf("%s\n", fname);
        }
        return EXIT_SUCCESS;
}
```

Besonders zu beachten sind die folgenden Punkte:

- Für den Parameter *destsize*von *snprintf()*wird der Wert durch den *sizeof*-Operator berechnet und nicht etwa direkt angegeben. Dadurch muss bei späteren Änderungen die 256 nur an genau einer Stelle verändert werden. Das beugt Fehlern vor.
- Wenn der Rückgabewert von *snprintf()*mindestens so groß wie *sizeof(fname)*ist, bedeutet das, dass 256 reguläre Zeichen *und*das abschließende Nullzeichen in den Puffer geschrieben werden sollten. Dafür ist aber kein Platz gewesen, und es wurde ein Teil des Strings abgeschnitten.

int sprintf(char *dest,const char *format,...)

Achtung! Da diese Funktion nicht die Länge prüft kann es zum Pufferüberlauf kommen!Deshalb sollte besser `snprintf`verwendet werden. Wenn dennoch diese Funktion genutzt wird, schreibt sie in den String *dest*den Formatstring *format*und gegebenenfalls die weiteren Parameter.

int vprintf(const char *format,va_list list)

Es wird zusätzlich der Header stdarg.h [107] benötigt !Die Funktion `vprintf`ist äquivalent zu der Funktion `printf`außer dass eine variable Argumentenliste anstelle des "..."-Operators genutzt wird. **Achtung diese Funktion ruft nicht das Makro va_end. Der Inhalt von *list*ist deshalb nach dem Funktionsaufruf undefiniert !**Nach der Funktion sollte die rufende Instanz deshalb unbedingt das Makro va_end als nächstes aufrufen.

int vfprintf(FILE *stream,const char *format,va_list list)

Es wird zusätzlich der Header stdarg.h [108] benötigt !Die Funktion `vfprintf`ist äquivalent zu der Funktion `fprintf`außer dass eine variable Argumentenliste anstelle des ...-Operators genutzt wird. **Achtung diese Funktion ruft nicht das Makro va_end. Der Inhalt von *list*ist deshalb nach dem Funktionsaufruf undefiniert!**Nach der Funktion sollte die rufende Instanz deshalb unbedingt das Makro va_end als nächstes aufrufen.

int vsprintf(char *dest,const char *format,va_list list)

Es wird zusätzlich der Header stdarg.h [109] benötigt !Die Funktion `vsprintf`ist äquivalent zu der Funktion `sprintf`außer dass eine variable Argumentenliste anstelle des "..."-Operators genutzt wird. **Achtung diese Funktion ruft nicht das Makro va_end. Der Inhalt von *list*ist deshalb nach dem Funktionsaufruf undefiniert !**Nach der Funktion sollte die rufende Instanz deshalb unbedingt das Makro va_end als nächstes aufrufen.

107 Kapitel 22.10.10 auf Seite 181
108 Kapitel 22.10.10 auf Seite 181
109 Kapitel 22.10.10 auf Seite 181

int vsnprintf(char *dest,size_t destsize,const char *format,va_list list)

Es wird zusätzlich der Header stdarg.h [110] benötigt !Die Funktion `vsnprintf`ist äquivalent zu der Funktion `snprintf`außer dass eine variable Argumentenliste anstelle des ''...''-Operators genutzt wird. **Achtung diese Funktion ruft nicht das Makro va_end. Der Inhalt von *list*ist deshalb nach dem Funktionsaufruf undefiniert!**Nach der Funktion sollte die rufende Instanz deshalb unbedingt das Makro va_end als nächstes aufrufen.

int scanf (const char *formatString, ...)

entspricht `fscanf(stdin, const char* formatString, ...)`

int fscanf(FILE *fp,const char *format,...)

Die Funktion `fscanf(FILE *fp,const char *format);`liest eine Eingabe aus dem mit *fp*übergebenen Stream. Über den Formatstring *format*wird `fscanf`mitgeteilt welchen Datentyp die einzelnen Elemente haben, die über Zeiger mit den Werten der Eingabe gefüllt werden. Der Rückgabewert der Funktion ist die Anzahl der erfolgreich eingelesen Datentypen bzw. im Fehlerfall der Wert der Konstante `EOF`.

int sscanf(const char *src,const *format,...)

Die Funktion ist äquivalent zu `fscanf`, außer dass die Eingabe aus dem String *src*gelesen wird.

int vscanf(const char *fromat,va_list list)

Es wird zusätzlich der Header stdarg.h [111] benötigt !Die Funktion `vscanf`ist äquivalent zu der Funktion `scanf`außer dass mit einer Argumentenliste gearbeitet wird. **Achtung es wird nicht das Makro *va_end*aufgerufen. Der Inhalt von *list*ist nach der Funktion undefiniert!**

int vsscanf(const char *src,const char *fromat,va_list list)

Es wird zusätzlich der Header stdarg.h [112] benötigt !Die Funktion `vsscanf`ist äquivalent zu der Funktion `sscanf`außer dass mit einer Argumentenliste gearbeitet wird. **Achtung es wird nicht das Makro *va_end*gerufen der Inhalt von *list*ist nach der Funktion undefiniert!**

110 Kapitel 22.10.10 auf Seite 181
111 Kapitel 22.10.10 auf Seite 181
112 Kapitel 22.10.10 auf Seite 181

int fgetc(FILE *stream)

Die Funktion `fgetc(stream);` liefert das nächste Zeichen im Stream oder im Fehlerfall `EOF`.

22.10.13 stdlib.h

Die Datei stdlib.h enthält Funktionen zur Umwandlung von Variablentypen, zur Erzeugung von Zufallszahlen, zur Speicherverwaltung, für den Zugriff auf die Systemumgebung, zum Suchen und Sortieren, sowie für die Integer-Arithmetik (z. B. die Funktion `abs`für den Absolutbetrag eines Integers).

`EXIT_SUCCESS`

Diese Macro enthält den Implementierungsspezifischen Rückgabewert für ein erfolgreich ausgeführtes Programm.

`EXIT_FAILURE`

Diese Macro enthält den Implentierungsspezifischen Rückgabewert für ein fehlerhaft beendetes Programm.

`NULL`

Das Macro repräsentiert ein Zeiger auf ein nicht gültigen Speicherbereich wie in stddef.h [113] erklärt wird.

`RAND_MAX`

Das Makro ist implementierungsabhängig und gibt den Maximalwert den die Funktion rand() [114] zurückgeben kann.

```
double atof (const char *nptr)
```

Wandelt den Anfangsteil (gültige Zahlendarstellung) einer Zeichenfolge, auf die `nptr`zeigt, in eine `double`-Zahl um.

```
int atoi (const char *nptr)
```

Wandelt den Anfangsteil (gültige Zahlendarstellung) einer Zeichenfolge, auf die `nptr`zeigt, in eine `int`-Zahl um.

113 Kapitel 22.10.11 auf Seite 182
114 Kapitel 22.10.14 auf Seite 191

```
long atol (const char *nptr)
```

Wandelt den Anfangsteil (gültige Zahlendarstellung) einer Zeichenfolge, auf die `nptr`zeigt, in eine `long int`-Zahl um.

```
void exit(int fehlercode)
```

Beendet das Programm

```
int rand(void)
```

Liefert eine Pseudo-Zufallszahl im Bereich von 0 bis `RAND_MAX` [115].

```
long int strtol(const restrict char* nptr, char** restrict endp, int base);
```

Die Funktion `strtol`(string to long) wandelt den Anfang einer Zeichenkette `nptr`in einen Wert des Typs `long int`um. In `endp`wird ein Zeiger in `*endp`auf den nicht umgewandelten Rest abgelegt, sofern das Argument ungleich `NULL`ist.

Die Basis legt fest, um welches Stellenwertsystem es sich handelt. (2 für das Dualsystem, 8 für das Oktalsystem, 16 für das Hexadezimalsystem und 10 für das Dezimalsystem). Die Basis kann Werte zwischen 2 und 36 sein. Die Buchstaben von a (bzw. A) bis z (bzw. Z) repräsentieren die Werte zwischen 10 und 35. Es sind nur Ziffern und Buchstaben erlaubt, die kleiner als `base`sind. Ist der Wert für `base`0, so wird entweder die Basis 8 (`nptr`beginnt mit 0), 10 (`nptr`beginnt mit einer von 0 verschiedenen Ziffer) oder 16 (`nptr`beginnt mit 0x oder 0X) verwendet. Ist die Basis 16, so zeigt eine vorangestelltes 0x bzw. 0X an, dass es sich um eine Hexadezimalzahl handelt. Wenn `nptr`mit einem Minuszeichen beginnt, ist der Rückgabewert negativ.

Ist die übergebene Zeichenkette leer oder hat nicht die erwartete Form, wird keine Konvertierung durchgeführt und 0 wird zurückgeliefert. Wenn der korrekte Wert größer als der darstellbare Wert ist, wird `LONG_MAX`zurückgegeben, ist er kleiner wird `LONG_MIN`zurückgegeben und das Makro `ERANGE`wird in `errno`abgelegt.

```
long long int strtoll(const restrict char* nptr, char** restrict endp, int
base);
```

(neu in C99 eingeführt)

Die Funktion entspricht `strtol`mit dem Unterschied, das der Anfang des Strings `nptr`in einen Wert des Typs `long long int`umgewandelt wird. Wenn der korrekte Wert größer als der darstellbare Wert ist, wird `LLONG_MAX`zurückgegeben, ist er kleiner, wird `LLONG_MIN`zurückgegeben.

115 Kapitel 22.10.14 auf Seite 191

```
unsigned long int strtoul(const restrict char* nptr, char** restrict endp,
int base);
```

Die Funktion entspricht `strtol`mit dem Unterschied, das der Anfang des Strings `nptr`in einen Wert des Typs `ungsigned long int`umgewandelt wird. Wenn der korrekte Wert größer als der darstellbare Wert ist, wird `ULONG_MAX`zurückgegeben.

```
unsigned long long int strtoull(const restrict char* nptr, char** restrict
endp, int base);
```

(neu in C99 eingeführt)

Die Funktion entspricht `strtol`mit dem Unterschied, das der Anfang des Strings `nptr`in einen Wert des Typs `ungsigned long long int`umgewandelt wird. Wenn der korrekte Wert größer als der darstellbare Wert ist, wird `ULLONG_MAX`zurückgegeben.

```
void* malloc(size_t size)
```

Die Funktion fordert vom System size byte an speicher an und gibt im Erfolgsfall einen Zeiger auf den Beginn des Bereiches zurück, im Fehlerfall NULL.

```
void free(void *ptr)
```

Gibt den dynamischen Speicher, der durch ptr repräsentiert wurde wieder frei.

```
int system(const char* command);
```

Führt den mit `command`angegebenen Befehl als Shell-Befehl aus und gibt den Rückgabewert des ausgeführten Prozesses zurück.

```
void qsort(void *base, size_t nmemb, size_t size, int(*compar)(const void
*, const void *));
```

Die `qsort`Funktion sortiert ein Array von Elementen mit dem Quicksort[116] Algorithmus. Der Parameter `base`ist ein Zeiger auf den Anfang des Arrays. `nmemb`gibt die Anzahl des Elemente in dem zu sortierenden Array an. Mit dem Parameter `size`übergibt man die Größe eines einzelnen Elements des Arrays. `compar`ist ein Zeiger auf eine Funktion, welche zwei Elemente nach dem gewünschten Sortierkriterium vergleicht.

116 `http://de.wikipedia.org/wiki/Quicksort`

22.10.14 string.h

Die Datei string.h enthält Funktionen zum Bearbeiten und Testen von Zeichenketten.

- `char* strcpy(char* sDest, const char* sSrc)`[117]Kopiert einen String sSrc nach sDest inklusive '\0'
- `char* strcat(char* s1, const char* s2)`[118]Verbindet zwei Zeichenketten miteinander
- `void* strncpy(char* sDest, const char* sSrc, size_t n)`[119]Wie `strcpy`, kopiert jedoch maximal n Zeichen (ggf. ohne '\0').
- `size_t strlen(const char* s)`[120]Liefert die Länge einer Zeichenkette ohne '\0'
- `int strcmp(const char* s1, char* s2)`[121]Vergleicht zwei Zeichenketten miteinander. Liefert 0, wenn s1 und s2 gleich sind, <0 wenn s1<s2 und >0 wenn s1>s2
- `int strstr(const char* s, const char* sSub)`[122]Sucht die Zeichenkette sSub innerhalb der Zeichenkette s. Liefert einen Zeiger auf das erste Auftreten von sSub in s, oder NULL, falls sSub nicht gefunden wurde.
- `int strchr(const char* s, int c)`[123]Sucht das erste Auftreten des Zeichens c in der Zeichenkette s. Liefert einen Zeiger auf das entsprechende Zeichen in s zurück, oder NULL, falls das Zeichen nicht gefunden wurde.
- `void* memcpy(void* sDest, const void* sSrc, size_t n)`[124]Kopiert n Bytes von sSrc nach sDest, liefert sDest. Die Speicherblöcke dürfen sich *nicht*überlappen.
- `void* memmove(void* sDest, const void* sSrc, size_t n)`[125]Kopiert n Bytes von sSrc nach sDest, liefert sDest. Die Speicherblöcke dürfen sich überlappen.

22.10.15 time.h

time.h enthält Kalender- und Zeitfunktionen.

- `time_t`Arithmetischer Typ, der die Kalenderzeit repräsentiert.
- `time_t time(time_t *tp)`Liefert die aktuelle Kalenderzeit. Kann keine Kalenderzeit geliefert werden, so wird der Wert –1 zurückgegeben. Als Übergabeparameter kann ein Zeiger übergeben werden, der nach Aufruf der Funktion ebenfalls die Kalenderzeit liefert. Bei NULL wird dem Zeiger kein Wert zugewiesen.

117 Kapitel 11.1 auf Seite 105
118 Kapitel 11.1 auf Seite 105
119 Kapitel 11.1 auf Seite 105
120 Kapitel 11.1 auf Seite 105
121 Kapitel 11.1 auf Seite 105
122 Kapitel 11.1 auf Seite 105
123 Kapitel 11.1 auf Seite 105
124 Kapitel 11.1 auf Seite 105
125 Kapitel 11.1 auf Seite 105

22.11 Neue Header in ISO C (C94/C95)

22.11.1 iso646.h

Folgende Makros sind im Header `<iso646.h>`definiert, die als alternative Schreibweise für die logischen Operatoren verwendet werden können:

Makro	**Operator**
and	&&
and_eq	&=
bitand	&
compl	~
not	!
not_eq	!=
or	||
or_eq	|=
xor	^
xor_eq	^=

22.11.2 wchar.h

- `int fwprintf(FILE *stream, const wchar_t *format, ...);`:

  wide character Variante von `fprintf`
- `int fwscanf(FILE *stream, const wchar_t *format, ...);`:

  wide character Variante von `fscanf`
- `wprinf(const wchar_t *format, ... );`:

  wide character Variante von `printf`
- `wscanf(const wchar_t *format, ...);`:

  wide character :Variante von `scanf`
- `wint_t getwchar(void);`:

  wide character Variante von `getchar`
- `wint_t putwchar(wchar_t c);`:

  wide character Variante von `putchar`
- `wchar_t *wcscpy(wchar_t *s1, const wchar_t *s2);`:

  wide character Variante von `strcpy`
- `wchar_t *wcscat(wchar_t *s1, const wchar_t *s2);`:

  wide character Variante von `strcat`
- `wchar_t *wcscmp(const wchar_t *s1, const wchar_t *s2);` _

wide character Variante von `strcmp`

- `size_t wcslen(const wchar_t *s);`:

  wide character Variante von `strlen`

22.11.3 wctype.h

22.12 Neue Header in ISO C (C99)

22.12.1 complex.h

Dieser Header definiert Macros und Funktionen um mit komplexen Zahlen[126] zu rechnen.

Realanteil bestimmen

- `double creal(double complex z);`
- `float crealf(float complex z);`
- `long double creall(long double complex z);`

Imaginäranteil bestimmen

- `double cimag(double complex z);`
- `float cimagf(float complex z);`
- `long double cimagl(long double complex z);`

Betrag einer komplexen Zahl bestimmen

- `double cabs(double complex z);`
- `float cabsf(float complex z);`
- `long double cabsl(long double complex z);`

Winkel φ bestimmen

- `double carg(double complex z);`
- `float cargf(float complex z);`
- `long double cargl(long double complex z);`

Komplexe Konjugation

- `double complex conj(double complex z);`
- `float complex conjf(float complex z);`
- `long double complex conjl(long double complex z);`

Wurzel

- `double complex csqrt(double complex z);`
- `float complex csqrtf(float complex z);`
- `long double complex csqrtl(long double complex z);`

Sinus

126 `http://de.wikipedia.org/wiki/Komplexe_Zahl`

- `double complex csin(double complex z);`
- `float complex csinf(float complex z);`
- `long double complex csinl(long double complex z);`

Kosinus

- `double complex ccos(double complex z);`
- `float complex ccosf(float complex z);`
- `long double complex ccosl(long double complex z);`

Tangens

- `double complex ctan(double complex z);`
- `float complex ctanf(float complex z);`
- `long double complex ctanl(long double complex z);`

Exponentialfunktion

- `double complex cpow(double complex x, complex double z);`
- `float complex cpowf(float complex x, complex float z);`
- `long double complex cpowl(long double complex x, complex long double z);`

natürliche Exponentialfunktion

- `double complex cexp(double complex z);`
- `float complex cexpf(float complex z);`
- `long double complex cexpl(long double complex z);`

natürlicher Logarithmus

- `double complex clog(double complex z);`
- `float complex clogf(float complex z);`
- `long double complex clogl(long double complex z);`

22.12.2 fenv.h

- `int feclearexcept(int excepts);`
- `int fegetexceptflag(fexcept_t *flagp, int excepts);`
- `int feraiseexcept(int excepts);`
- `int fesetexceptflag(const fexcept_t *flagp, int excepts);`
- `int fetestexcept(int excepts);`
- `int fegetround(void);`
- `int fesetround(int rounding_mode);`
- `int fegetenv(fenv_t *envp);`
- `int feholdexcept(fenv_t *envp);`
- `int fesetenv(const fenv_t *envp);`
- `int feupdateenv(const fenv_t *envp);`

22.12.3 inttypes.h

22.12.4 stdbool.h

Definiert den logischen Typ `bool`für die Verwendung.

22.12.5 stdint.h

22.12.6 tgmath.h

22.13 Anweisungen

Diese Darstellungen stützen sich auf die Sprache C gemäß ISO/IEC 9899:1999 (C99). Auf Dinge, die mit C99 neu eingeführt wurden, wird im Folgenden gesondert hingewiesen.

Anweisungen und Blöcke sind Thema des Kapitels *6.8 Statements and blocks*in C99.

22.13.1 Benannte Anweisung

Benannte Anweisungen sind Thema des Kapitels *6.8.1 Labeled statements*in C99.

Syntax:

```
Bezeichner : Anweisung
case konstanter Ausdruck : Anweisung
default : Anweisung
```

Sowohl die `case`-Anweisung, wie auch die `default`-Anweisung dürfen nur in einer `switch`-Anweisung verwendet werden.

Siehe auch:`switch`[127].

Der Bezeichner der benannten Anweisung kann in der gesamten Funktion angesprochen werden. Sein Gültigkeitsbereich ist die Funktion. Dies bedeutet, dass der Bezeichner in einer `goto`-Anweisung noch vor seiner Deklaration verwendet werden kann.

Siehe auch:`goto`[128], `if`[129]

22.13.2 Zusammengesetzte Anweisung

Das Kapitel *6.8.2 Compound statement*in C99 hat die zusammengesetzten Anweisungen zum Thema.

Syntax (C89):

127 Kapitel 22.13.4 auf Seite 201
128 Kapitel 22.13.6 auf Seite 207
129 Kapitel 22.13.4 auf Seite 199

```
{ declaration-list_opt statement-list_opt }
```

declaration-list:

```
Deklaration
declaration-list Deklaration
```

statement-list:

```
Anweisung
statement-list Anweisung
```

Syntax (C99):

```
{ block-item-list_opt }
```

block-item-list:

```
block-item
block-item-list block-item
```

block-item:

```
Deklaration
Anweisung
```

Eine zusammengesetzte Anweisung bildet einen *Block*[130].

Zusammengesetzte Anweisungen dienen dazu, mehrere Anweisungen zu einer einzigen Anweisung zusammenzufassen. So verlangen viele Anweisungen *eine*Anweisung als Unteranweisung. Sollen jedoch mehrere Anweisungen als Unteranweisung angegeben werden, so steht oft nur der Weg zur Verfügung, diese Anweisungen als eine Anweisung zusammenzufassen.

Wesentliches Merkmal der Syntax zusammengesetzter Anweisungen sind die umschließenden geschweiften Klammern (`{}`). Bei Anweisungen, die Unteranweisungen erwarten, wie beispielsweise Schleifen oder Verzweigungen, werden geschweifte Klammern so häufig eingesetzt, dass leicht der falsche Eindruck entsteht, sie seien Teil der Syntax der Anweisung. Lediglich die Syntax einer Funktionsdefinition verlangt (in C99) die Verwendung einer zusammengesetzen Anweisung.

Beispiel:

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

130 Kapitel 22.14.2 auf Seite 212

```
int main(int argc, char *argv[])
{
        if (argc > 1)
                printf("Es wurden %d Parameter angegeben.\n", argc-1);
                printf("Der erste Parameter ist '%s'.\n", argv[1]);

        return 0;
}
```

Das eben gezeigte Beispiel lässt sich übersetzen, jedoch ist sein Verhalten nicht das Gewünschte. Der erste Parameter der Applikation soll nur ausgegeben werden, wenn er angegeben wurde. Jedoch wird nur die erste `printf`-Anweisung (eine Ausdrucksanweisung) bedingt ausgeführt. Die zweite `printf`-Anweisung wird *stets*ausgeführt, auch wenn die Formatierung des Quelltextes einen anderen Eindruck vermittelt. Repräsentiert der Ausdruck `argv[1]`keinen gültigen Zeiger, so führt seine Verwendung beim Aufruf der Funktion `printf`zu einem undefinierten Verhalten der Applikation.

Siehe auch:`if`[131]

Es soll also auch der zweite Aufruf von `printf`nur dann erfolgen, wenn mindestens ein Parameter angegeben wurde. Dies kann erreicht werden, indem beide (Ausdrucks-)Anweisungen zu einer Anweisung zusammengesetzt werden. So arbeitet das folgende Beispiel wie gewünscht.

Beispiel:

```
#include <stdio.h>

int main(int argc, char *argv[])
{
        if (argc > 1)
        {
                printf("Es wurden %d Parameter angegeben.\n", argc-1);
                printf("Der erste Parameter ist '%s'.\n", argv[1]);
        }

        return 0;
}
```

In zusammengesetzten Anweisungen können neue Bezeichner deklariert werden. Diese Bezeichner gelten ab dem Zeitpunkt ihrer Deklaration und bis zum Ende des sie umschließenden Blocks. Die wesentliche Änderung von C89 zu C99 ist, dass in C89 *alle*Deklarationen **vor***allen*Anweisungen stehen mussten. Im aktuellen Standard C99 ist dieser Zwang aufgehoben.

22.13.3 Ausdrucksanweisung

Ausdrucksanweisungen werden im Kapitel *6.8.3 Expression and null statements*in C99 beschrieben.

Syntax:

131 Kapitel 22.13.4 auf Seite 199

$Ausdruck_{\text{opt}}$;

Der Ausdruck einer Ausdrucksanweisung wird als `void`-Ausdruck und wegen seiner Nebeneffekte ausgewertet. Alle Nebeneffekte des Ausdrucks sind zum Ende der Anweisung abgeschossen.

Beispiel:

```
#include <stdio.h>

int main(int argc, char *argv[])
{
        int a = 1, b = 2, c;             /* Deklarationen */

        c = a + b;                       /* Ausdrucksanweisung */
        printf("%d + %d = %d\n",a,b,c);  /* Ausdrucksanweisung */

        return 0;                        /* Sprung-Anweisung */
}
```

In der ersten Ausdrucksanweisung `c = a + b;`wird der Ausdruck `c = a + b`mit seinem Teilausdruck `a + b`ausgewertet. Als Nebeneffekt wird die Summe aus den Werten in `a`und `b`gebildet und in dem Objekt `c`gespeichert. Der Ausdruck der zweiten Ausdrucksanweisung ist der Aufruf der Funktion `printf`. Als Nebeneffekt gibt diese Funktion einen Text auf der Standardausgabe aus. Häufige Ausdrucksanweisungen sind Zuweisungen und Funktionsaufrufe.

Leere Anweisung

Wird der Ausdruck in der Ausdrucksanweisung weggelassen, so wird von einer *leeren Anweisung*gesprochen. Leere Anweisungen werden verwendet, wenn die Syntax der Sprache C eine Anweisung verlangt, jedoch keine gewünscht ist.

Beispiel:

```
void foo (char * sz)
{
        if (!sz) goto ende;

        /* ... */
        while(getchar() != '\n' && !feof(stdin))
                ;

        /* ... */
        ende: ;
}
```

22.13.4 Verzweigungen

Die Auswahl-Anweisungen werden in C99 im Kapitel *6.8.4 Selection statements*beschrieben.

if

Syntax:

```
if (Ausdruck) Anweisung
if (Ausdruck) Anweisung else Anweisung
```

In beiden Formen der `if`-Anweisung muss der Kontroll-Ausdruck von einem skalaren Datentypen sein. Bei beiden Formen wird die erste Unteranweisung nur dann ausgeführt, wenn der Wert des Ausdruckes ungleich `0`(null) ergibt. In der zweiten Form der `if`-Anweisung wird die Unteranweisung nach dem Schlüsselwort `else`nur dann ausgeführt, wenn der Kontrollausdruck den Wert `0`darstellt. Wird die erste Unteranweisung über einen Sprung zu einer benannten Anweisung[132] erreicht, so wird die Unteranweisung im `else`-Zweig nicht ausgeführt. Das Schlüsselwort `else`wird stets jenem `if`zugeordnet, das vor der *vorangegangenen*Anweisung steht.

Beispiel:

```
#include <stdio.h>

int main(int argc, char *argv[])
{
        if (argc > 1)
                printf("Erster Parameter: %s\n", argv[1]);

        return 0;
}
```

Im vorangegangenen Beispiel prüft das Programm, ob mindestens ein Parameter dem Programm übergeben wurde und gibt ggf. diesen Parameter aus. Wurde jedoch *kein*Parameter dem Programm übergeben, so wird die Ausdrucksanweisung `printf("Erstes Argument: %s\n", argv[1]);`*nicht*ausgeführt.

Soll auch auf den Fall eingegangen werden, dass der Kontrollausdruck `0`ergeben hatte, so kann die zweite Form der `if`-Anweisung verwendet werden. Im folgenden Beispiel genau eine der beiden Unteranweisungen ausgeführt.

Beispiel:

```
#include <stdio.h>

int main(int argc, char *argv[])
{
        if(argc > 1)
                printf("Erster Parameter: %s\n", argv[1]);
        else
                puts("Es wurde kein Parameter übergeben.");

        return 0;
}
```

Die `if`-Anweisung stellt, wie der Name schon sagt, *eine Anweisung*dar. Daher kann eine `if`-Anweisung ebenso als Unteranweisung einer anderen Anweisung verwendet werden. Wird

132 Kapitel 22.13.1 auf Seite 195

eine `if`-Anweisung als Unteranweisung einer anderen `if`-Anweisung verwendet, so ist darauf zu achten, dass sich ein eventuell vorhandenes `else`stets an das voranstehende `if`bindet.

Beispiel:

```
#include <stdio.h>

int main(int argc, char *argv[])
{
        if (argc > 1)
                if (argc == 2)
                        puts("Es wurde genau ein Parameter übergeben.");
        else
                puts("Es wurde kein Parameter übergeben.");

        return 0;
}
```

Die Formatierung des Quelltextes im letzten Beispiel erweckt den Eindruck, dass der Text `Es wurde kein Argument übergeben.`nur dann ausgegeben wird, wenn der Ausdruck `argc > 1`den Wert `0`ergibt. Jedoch ist Ausgabe des Textes davon abhängig, ob der Ausdruck `argc == 2`den Wert `0`ergibt. Somit wird beim Fehlen eines Parameters *kein*Text ausgegeben und im Fall von mehreren Parametern erhalten wir die Fehlinformation, dass keine Parameter übergeben worden seien.

Soll das gewünschte Verhalten erreicht werden, so kann die `if`-Anweisung, welche die erste Unteranweisung darstellt, durch eine zusammengesetzte Anweisung[133] ersetzt werden. Noch einmal auf deutsch: sie kann geklammert werden. So findet im folgenden Beispiel das Schlüsselwort `else`vor sich eine zusammengesetzte Anweisung und vor dieser Anweisung jenes `if`, welches mit dem Kontrollausdruck `argc > 1`behaftet ist.

Beispiel:

```
#include <stdio.h>

int main(int argc, char *argv[])
{
        if (argc > 1)
        {
                if(argc == 2)
                        puts("Es wurde genau ein Parameter übergeben.");
        } else
                puts("Es wurde kein Parameter übergeben.");

        return 0;
}
```

Ebenso wird die zusammengesetzte Anweisung verwendet, wenn mehrere Anweisungen bedingt ausgeführt werden sollen. Da die `if`-Anweisung stets nur *eine*Anweisung als Unteranweisung erwartet, können zum bedingten Ausführen mehrerer Anweisungen, diese wiederum geklammert werden.

Im nächsten Beispiel findet das letzte `else`als erste Unteranweisung eine `if`-Anweisung mit einem `else`-Zweig vor. Diese (Unter-)Anweisung ist abgeschlossen und kann nicht mehr erweitert werden. Daher kann sich an eine solche `if`-Anweisung das letzte `else`nicht binden.

133 Kapitel 22.13.2 auf Seite 195

Es gibt keine Form der If-Anweisung mit *zwei*Else-Zweigen. Somit arbeitet das folgende Programm auch ohne Klammern wie erwartet.

Beispiel:

```
#include <stdio.h>

int main(int argc, char *argv[])
{
        if (argc > 1)
                if (argc == 2)
                        puts("Es wurde genau ein Parameter übergeben.");
                else
                        puts("Es wurden mehr als ein Parameter übergeben.");
        else
                puts("Es wurde kein Argument übergeben.");

        return 0;
}
```

Im letzten Beispiel zum Thema der `if`-Anweisung soll noch gezeigt werden, wie sich ein Programm verhält, bei dem in eine Unteranweisung über einen Sprung aufgerufen wird.

Beispiel:

```
#include <stdio.h>

int main(void)
{
        goto marke;

        if (1==2)
                marke: puts("Ich werde ausgegeben.");
        else
                puts("Ich werde nicht ausgegeben!");

        return 0;
}
```

Obwohl der Ausdruck `1==2`offensichtlich den Wert `0`liefert, wird der `else`-Zweig *nicht*ausgeführt.

switch

Die `switch`-Anweisung wird im Kapitel *6.8.4.2 The switch statement*in C99 besprochen.

Syntax:

```
switch (Ausdruck) Anweisung
```

Die `switch`-Anweisung erlaubt eine Verzweigung mit mehreren Sprungzielen. Ihr Vorteil gegenüber der `if`-Anweisung[134] ist eine bessere Übersicht über den Programmfluss.

134 Kapitel 22.13.4 auf Seite 199

Der Ausdruck muss einen ganzzahligen Datentyp haben. Er ist der *Kontrollausdruck*. Der ganzzahlige Datentyp des Kontrollausdrucks ist eine Einschränkung gegenüber der `if`-Anweisung[135], die in dem Bedingungs-Ausdruck einen skalaren Datentyp verlangt.

Siehe auch:`if`[136]

Der Wert des Kontrollausdrucks bestimmt, zu welcher `case`-Anweisung[137] einer *zugehörigen Anweisung*gesprungen wird. In fast allen Fällen ist die zugehörige Anweisung eine zusammengesetzte Anweisung (`{}`), welche die verschiedenen `case`-Anweisungen aufnimmt.

Beispiel:

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

int main(int argc, char *argv[])
{
        int a=0, b=0;

        switch(argc)
                case 0: puts("Kein Programmname verfügbar.");

        switch(argc)
        {
                case 3: b = atoi (argv[2]);
                case 2: a = atoi (argv[1]);
                        printf("%d + %d = %d\n.", a, b, a+b);
        }

        return 0;
}
```

Im letzten Beispiel werden zwei `switch`-Anweisungen gezeigt. Die erste `switch`-Anweisung zeigt, dass geschweifte Klammern nach dem Standard C99 nicht zwangsläufig notwendig sind. Jedoch wird in einem solchen Fall wohl eher eine `if`-Anweisung verwendet werden. In der zweiten Ausgabe von *Programmieren in C*werden die geschweiften Klammern bei der `switch`-Anweisung noch verlangt.

Eine `case`-Anweisung bildet ein mögliches Sprungziel, bei dem die Programmausführung fortgesetzt wird. Die zweite `switch`-Anweisung im letzten Beispiel definiert zwei `case`-Anweisungen. Es werden zwei Sprungziele für den Fall definiert, dass `argc`den Wert `3`bzw. den Wert `2`hat. Der Ausdruck von `case`muss eine Konstante sein. Dabei darf *der Wert*des Ausdruck nicht doppelt vorkommen.

```
switch(var)
{
        case 2+3: ;
        case 1+4: ; /* illegal */
}
```

Hat ein `case`-Ausdruck einen anderen Typen als der Kontrollausdruck, so wird der Wert des `case`-Ausdruckes in den Typen des Kontrollausdruckes gewandelt.

135 Kapitel 22.13.4 auf Seite 199
136 Kapitel 22.13.4 auf Seite 199
137 Kapitel 22.13.1 auf Seite 195

Wird zu einem der beiden `case`-Anweisungen gesprungen, so werden auch alle nachfolgenden Anweisungen ausgeführt. Soll die Abarbeitung innerhalb der zugehörigen Anweisung abgebrochen werden, so kann die `break`[138]-Anweisung eingesetzt werden. So verhindert die Anweisung `break;`im nachfolgenden Beispiel, dass *beide*Meldungen ausgegeben werden, wenn kein Parameter beim Programmaufruf angegeben worden sein sollte. Jedoch fehlt eine Anweisung `break;`zwischen `case -1:`und `case 0:`. Dies hat zur Folge, dass in beiden Fällen die Meldung *Kein Parameter angegeben*ausgegeben wird. Der Ausdruck `argc - 1`nimmt den Wert `-1`an, wenn auch kein Programmname verfügbar ist.

Beispiel:

```
#include <stdio.h>

int main (int argc, char *argv[])
{
        switch (argc - 1)
        {
                case -1: case 0: puts("Kein Parameter angegeben");
                        break;
                case 1: puts("Ein Parameter angegeben.");
        }
        return 0;
}
```

Siehe auch:`break`[139]

Ergibt der Bedingungsausdruck einen Wert, zu dem es keinen entsprechenden Wert in einer Case-Anweisung gibt, so wird auch in keinen `case`-Zweig der zugehörigen Anweisung von `switch`gesprungen. Für diesen Fall kann eine Anweisung mit der Marke `default:`benannt werden. Bei der so benannten Anweisung wird die Programmausführung fortgesetzt, wenn kein `case`-Zweig als Sprungziel gewählt werden konnte. Es darf jeweils nur eine Marke `default`in einer `switch`-Anweisung angeben werden.

Um Bereiche abzudecken kann man auch "..." schreiben.

```
#include <stdio.h>

char puffer[256];
int main()
{
        printf("Geben Sie bitte Ihren Nachnamen ein.");
        fgets(puffer,256,stdin);
        switch(puffer[0])
        {
                case 'A'...'M':
                        printf("Sie stehen in der ersten Haelfte des
 Telefonbuches.");
                break;
                case 'N'...'Z':
                        printf("Sie stehen in der zweiten Haelfte des
 Telefonbuches.");
                break;
        }

        return 0;
}
```

138 Kapitel 22.13.6 auf Seite 210
139 Kapitel 22.13.6 auf Seite 210

Siehe auch: Benannte Anweisung[140]

22.13.5 Schleifen

Schleifen (Iterations-Anweisungen) werden im Kapitel *6.8.5 Iteration statements*in C99 beschrieben.

while

Die `while`-Anweisung ist Thema des Kapitels *6.8.5.1 The while statement*in C99.

Syntax:

```
while (Ausdruck) Anweisung
```

Der *Kontrollausdruck*muss von einem skalaren Datentypen sein. Er wird *vor*einer eventuellen Ausführung der zugehörigen Anweisung (Schleifenrumpf) ausgewertet. Der Schleifenrumpf wird ausgeführt, wenn der Kontrollausdruck einen Wert ungleich `0`ergeben hatte. Nach jeder Ausführung des Schleifenrumpfs wird Kontrollausdruck erneut ausgewertet um zu prüfen, ob mit der Abarbeitung des Schleifenrumpfs fortgefahren werden soll. Erst wenn der Kontrollausdruck den Wert `0`ergibt, wird die Abarbeitung abgebrochen. Ergibt der Kontrollausdruck schon bei der ersten Auswertung den Wert `0`, so wird der Schleifenrumpf überhaupt nicht ausgeführt. Die `while`-Anweisung ist eine *kopfgesteuerte Schleife*.

Sowohl die `while`-Anweisung, wie auch deren zugehörige Anweisung (Schleifenrumpf) bilden je einen *Block*[141].

Beispiel:

```
#include <stdio.h>

int main(int argc, char *argv[])
{
        while (argc > 0 && *++argv)
                puts(*argv);

        return 0;
}
```

Siehe auch:`do`[142], `for`[143], Block[144]

do

Die `do`-Anweisung wird im Kapitel *6.8.5.2 The do statement*in C99 beschrieben.

Syntax:

140 Kapitel 22.13.1 auf Seite 195
141 Kapitel 22.14.2 auf Seite 212
142 Kapitel 22.13.5 auf Seite 204
143 Kapitel 22.13.5 auf Seite 205
144 Kapitel 22.14.2 auf Seite 212

```
do Anweisung while (Ausdruck);
```

Im Unterschied zur `while`-Anweisung wertet die `do`-Anweisung den Kontrollausdruck erst *nach*der Ausführung einer zugehörigen Anweisung (Schleifenrumpf) aus. Die Ausführung des Schleifenrumpfs wird solange wiederholt, bis der Kontrollausdruck den Wert `0`ergibt. Dadurch, dass der Kontrollausdruck erst nach der Ausführung des Schleifenrumpfes ausgewertet wird, ist mindestens eine einmalige Ausführung des Schleifenrumpfes garantiert. Die `do`-Anweisung ist eine *fußgesteuerte Schleife*.

Sowohl die `do`-Anweisung, wie auch deren zugehörige Anweisung (Schleifenrumpf) bilden je einen *Block*[145].

Siehe auch:`while`[146], `for`[147], Block[148]

Neben der klassischen Rolle einer fußgesteuerten Schleife bietet sich die `do`-Anweisung an, wenn ein Makro mit mehreren Anweisungen geschrieben werden soll. Dabei soll das Makro der Anforderung genügen, wie *eine Anweisung*, also wie im folgenden Codefragment verwendet werden zu können.

```
#include <stdio.h>
#define  makro(sz) do { puts(sz); exit(0); } while(0)

        /* ... */
        if (bedingung)
                makro("Die Bedingung trifft zu");
        else
                puts("Die Bedingung trifft nicht zu");
        /* ... */
```

Bei der Definition des Makros fehlt das abschließende Semikolon, das in der Syntax der `do`-Anweisung verlangt wird. Dieses Semikolon wird bei der Verwendung des Makros (in der `if`-Anweisung) angegeben. Die `do`-Anweisung ist *eine Anweisung*und da der Kontrollausdruck bei der Auswertung den (konstanten) Wert `0`ergibt, wird der Schleifenrumpf (zusammengesetzte Anweisung[149]) genau *einmal*ausgeführt. Zu diesem Thema äußert sich auch das Kapitel 6.3 der FAQ von dclc[150].

for

Die `for`-Anweisung ist Thema des Kapitels *6.8.5.3 The for statement*in C99.

Syntax:

for (*Ausdruck-1*$_{\text{opt}}$; *Ausdruck-2*$_{\text{opt}}$; *Ausdruck-3*$_{\text{opt}}$) *Anweisung*

145 Kapitel 22.14.2 auf Seite 212
146 Kapitel 22.13.5 auf Seite 204
147 Kapitel 22.13.5 auf Seite 205
148 Kapitel 22.14.2 auf Seite 212
149 Kapitel 22.13.2 auf Seite 195
150 `http://www.dclc-faq.de/kap6.htm#6.3`

Syntax (C99):

for (*Ausdruck-1*$_{\text{opt}}$; *Ausdruck-2*$_{\text{opt}}$; *Ausdruck-3*$_{\text{opt}}$) *Anweisung*
for (*Deklaration Ausdruck-2*$_{\text{opt}}$; *Ausdruck-3*$_{\text{opt}}$) *Anweisung*

Die `for`-Anweisung (erste Form bei C99) verlangt bis zu drei Ausdrücke. Alle drei Ausdrücke sind optional. Werden die Ausdrücke *Ausdruck-1*oder *Ausdruck-3*angegeben, so werden sie als `void`-Ausdrücke ausgewertet. Ihre Werte haben also keine weitere Bedeutung. Der Wert von *Ausdruck-2*stellt hingegen den *Kontrollausdruck*dar. Wird dieser ausgelassen, so wird ein konstanter Wert ungleich `0`angenommen. Dies stellt eine *Endlosschleife*dar, die nur durch eine Sprunganweisung[151] verlassen werden kann.

Zu Beginn der `for`-Anweisung wird der *Ausdruck-1*ausgewertet. Der Wert von *Ausdruck-2*muss von einem skalaren Datentypen sein und sein Wert wird *vor*einer eventuellen Ausführung der zugehörigen Anweisung (Schleifenrumpf) ausgewertet. Nach einer Ausführung des Schleifenrumpfs wird der *Ausdruck-3*ausgewertet. Wird der Schleifenrumpf nicht ausgeführt, so wird auch nicht der *Ausdruck-3*ausgewertet.

Beispiel:

```
#include <stdio.h>

int main(int argc, char *argv[])
{
        int i;

        for (i = 0; i < 10; ++i)
                printf("In der for-Anweisung: i = %2d\n", i);

        printf("Nach der for-Anweisung: i = %d\n\n", i);

        i = 0;
        while ( i < 10)
        {
                printf("In der while-Anweisung: i = %2d\n", i);
                ++i;
        }

        printf("Nach der while-Anweisung: i = %d\n", i);

        return 0;
}
```

Das letzte Beispiel zeigt, wie eine `for`-Anweisung durch eine `while`-Anweisung ersetzt werden könnte. Es soll nicht unerwähnt bleiben, dass die beiden Formen *nicht*das Selbe darstellen. Zum Einen stellt die `for`-Anweisung *eine Anweisung*dar, während in dem Beispiel die `while`-Anweisung von einer Ausdrucksanweisung[152] begleitet wurde. Würden wir beide Anweisungen zusammenfassen[153], so würden wir einen Block[154] mehr definieren.

Sowohl die `for`-Anweisung, wie auch deren Schleifenrumpf bilden je einen *Block*[155].

151 Kapitel 22.13.5 auf Seite 204
152 Kapitel 22.13.3 auf Seite 197
153 Kapitel 22.13.2 auf Seite 195
154 Kapitel 22.14.2 auf Seite 212
155 Kapitel 22.14.2 auf Seite 212

Auch wenn alle drei Ausdrücke optional sind, so sind es die Semikola (`;`) nicht. Die Semikola müssen *alle* angegeben werden.

Mit C99 ist die Möglichkeit eingeführt worden, eine Deklaration angeben zu können. Dabei ersetzt die Deklaration den *Ausdruck-1* **und** das *erste* Semikolon. Bei der Angabe der Definition wurde keineswegs die Angabe eines Semikolons zwischen der *Deklaration* und dem *Ausdruck-2* vergessen. In dieser Form ist die Angabe der *Deklaration* **nicht** optional, sie muss also angegeben werden. Eine Deklaration wird *immer* mit einem Semikolon abgeschlossen. Wird diese Form der `for`-Anweisung verwendet, so ergibt sich das oben fehlende Semikolon durch die Angabe einer Deklaration.

Der Geltungsbereich der mit *Deklaration* deklarierten Bezeichner umfasst sowohl *Ausdruck-2*, *Ausdruck-3* wie auch *Anweisung*, dem Schleifenrumpf. Der Bezeichner steht auch innerhalb von *Deklaration* zur Verfügung, soweit dies nach der Syntax von Deklarationen in C definiert ist.

Beispiel:

```
#include <stdio.h>

int main(int argc, char *argv[])
{
        for (int i = 0; i < 10; ++i)
        {
                int j = 0;
                printf("i = %2d, j = %d\n", i, j);
                ++j;
        }

        /* i und j sind hier nicht verfügbar */

        return 0;
}
```

Siehe auch: `while`[156], `do`[157], Block[158]

22.13.6 Sprunganweisungen

Die Sprunganweisungen werden im Kapitel *6.8.6 Jump statements* von C99 besprochen. Sie haben einen (bedingungslosen) Sprung zu einer anderen Stelle im Programm zur Folge.

goto

Die `goto`-Anweisung ist Thema des Kapitels *6.8.6.1 The goto statement* in C99.

Syntax:

goto *Bezeichner* ;

156 Kapitel 22.13.5 auf Seite 204
157 Kapitel 22.13.5 auf Seite 204
158 Kapitel 22.14.2 auf Seite 212

Mit der goto-Anweisung kann die Programmausführung bei einer benannten Anweisung[159] fortgesetzt werden. Dabei muss die benannte Anweisung in der *gleichen Funktion*angegeben worden sein. Da der Name der Anweisung in der gesamten Funktion gültig ist, kann auch „nach vorne“ gesprungen werden.

Siehe auch: Benannte Anweisung[160]

Bei einem Sprung in den Geltungsbereich eines Bezeichners darf dieser Bezeichner *nicht*ein *Feld mit variabler Länge*bezeichnen. Entsprechend der Regeln für einen Block[161] werden die Objekte angelegt, diese dürfen jedoch nicht initialisiert worden sein, da die Deklaration übersprungen wurde.

Beispiel:

```
/* Vorsicht! undefiniertes Verhalten */
#include <stdio.h>

int main (void)
{
        goto weiter;
        {
                int i = 99;
                weiter:
                printf("i = %d\n", i); /* i ist nicht initialisiert! */
        }
        return 0;
}
```

In dem letzten Beispiel ist das Objekt iin der Funktion printfnicht initialisiert. Daher ist das Verhalten des Programms undefiniert.

Siehe auch: Block[162]

Die Verwendung der goto-Anweisung hat gelegentlich einen, für den Menschen schwer lesbaren Programmcode zur Folge. Wie hoch die Lesbarkeit eines Quelltextes für einen Menschen als Qualitätsmerkmal für ein Programm bewertet wird, dass für einen Rechner geschrieben wurde, ist individuell verschieden. Dennoch soll nicht verschwiegen werden, dass die goto-Anweisung einen schlechten Ruf besitzt und viele Programmierer von ihrer Verwendung abraten. Dieser sollte jedoch nicht dazu führen, dass auf die goto-Anweisung verzichtet wird, obwohl ihre Verwendung eine einfachere Programmstruktur zur Folge gehabt hätte.

continue

Syntax:

```
continue ;
```

159 Kapitel 22.13.1 auf Seite 195
160 Kapitel 22.13.1 auf Seite 195
161 Kapitel 22.14.2 auf Seite 212
162 Kapitel 22.14.2 auf Seite 212

Die Anweisung `continue;`darf nur in den Schleifenanweisungen[163] `while`[164], `do`[165] und `for`[166] verwendet werden. Sie wird im oder als Schleifenrumpf angegeben. Die `continue`-Anweisung bricht die Abarbeitung des Schleifenrumpfs ab und prüft die Bedingungsanweisung der Schleife erneut. So ist in den folgenden Quelltextfragmenten die `continue`-Anweisung mit der Anweisung `goto weiter;`austauschbar.

Hinweis: Die Sprungmarke `weiter:`ist für die Funktion der `continue`-Anweisung *nicht*erforderlich.

```
for (int i = 0; i < 10; ++i)
{
        /* ... */
        continue;
        /* ... */
        weiter: ;
}


int i = 0;
while (i < 10)
{
        /* ... */
        continue;
        /* ... */
        ++i;
        weiter: ;
}


int i = 0;
do
{
        /* ... */
        continue;
        /* ... */
        ++i;
        weiter: ;
} while (i < 10);
```

Das folgende Beispiel gibt eine Folge von Multiplikationen aus. Dabei wird jedoch die Multiplikation ausgelassen, bei der das Ergebnis eine `10`ergeben hat.

Beispiel:

```
#include <stdio.h>

int main (void)
{
        for(int i = 0; i < 10; ++i)
        {
                int erg = 2 * i;

                if(erg == 10) continue;

                printf("2 * %2d = %2d\n", i, erg);
        }
```

163 Kapitel 22.13.5 auf Seite 204
164 Kapitel 22.13.5 auf Seite 204
165 Kapitel 22.13.5 auf Seite 204
166 Kapitel 22.13.5 auf Seite 205

```
        return 0;
}
```

Siehe auch:`break`[167], Schleifen[168]

break

Die `break`-Anweisung ist Thema der Kapitels *6.8.6.3 The break statement*in C99.

Syntax:

break ;

Die Anweisung `break;`bricht die for[169], `do`[170], `while`[171] oder `switch`[172]-Anweisung ab, die der `break`-Anweisung am nächsten ist. Bei einer for[173]-Anweisung wird nach einer `break`-Anweisung der *Ausdruck-3*nicht mehr ausgewertet.

Beispiel:

```
#include <stdio.h>

int main (void)
{
        int i;
        for(i = 0; i < 10; ++i)
                if ( i == 5)
                        break;

        printf("i = %d\n", i);

        return 0;
}
```

Das Programm im letzten Beispiel gibt eine `5`auf der Standardausgabe aus.

return

Die Anweisung `return;`wird in dem Kapitel *6.8.6.4 The return statement*in C99 beschrieben.

Syntax:

return $Ausdruck_{\text{opt}}$;

167 Kapitel 22.13.6 auf Seite 210
168 Kapitel 22.13.5 auf Seite 204
169 Kapitel 22.13.5 auf Seite 205
170 Kapitel 22.13.5 auf Seite 204
171 Kapitel 22.13.5 auf Seite 204
172 Kapitel 22.13.4 auf Seite 201
173 Kapitel 22.13.5 auf Seite 205

Die `return`-Anweisung beendet die Abarbeitung der aktuellen Funktion. Wenn eine `return`-Anweisung mit einem *Ausdruck*angegeben wird, dann wird der Wert des Ausdrucks an die aufrufende Funktion als Rückgabewert geliefert. In einer Funktion können beliebig viele `return`-Anweisungen angegeben werden. Jedoch muss dabei darauf geachtet werden, dass nachfolgender Programmcode durch den Programmfluss noch erreichbar bleibt.

Beispiel:

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

int IsFred(const char *sz)
{
        if (!sz)
                return 0;
        if (!strcmp(sz, "Fred Feuerstein"))
                return 1;
        return 0;
        puts("Dies wird nie ausgeführt.");
}
```

Der *Ausdruck*in der `return`-Anweisung ist optional. Dennoch gelten für ihn besondere Regeln. So darf der *Ausdruck*in Funktionen vom Typ `void`nicht angegeben werden. Ebenso darf der *Ausdruck*nur in Funktionen vom Typ `void`weggelassen werden.

Hat der *Ausdruck*der `return`-Anweisung einen anderen Typ als die Funktion, so wird der Wert des Ausdruckes in den Typ gewandelt, den die Funktion hat. Dies geschieht nach den gleichen Regeln, wie bei einer Zuweisung in ein Objekt vom gleichen Typen wie die Funktion.

22.14 Begriffserklärungen

Die Begriffe in diesem Abschnitt werden im Kapitel *6.8 Statements and blocks*in C99 erklärt.

22.14.1 Anweisung

Eine Anweisung ist eine Aktion, die ausgeführt wird.

Eine Anweisung wird in einer Sequenz (sequence point) ausgeführt. Jedoch kann eine Anweisung in mehrere Sequenzen aufgeteilt sein.

Gelegentlich ist die Aussage zu lesen, dass in der Sprache C *alle*Anweisungen mit einem Semikolon abgeschlossen werden. Dies ist nicht richtig. Lediglich die Ausdrucksanweisung[174], die do[175]-Anweisung und die Sprung-Anweisungen[176] werden mit einem Semikolon abgeschlossen. So verwendet folgendes Programm *kein*Semikolon.

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

174 Kapitel 22.13.3 auf Seite 197
175 Kapitel 22.13.5 auf Seite 204
176 Kapitel 22.13.5 auf Seite 204

```
int main (void)
{
        if (printf("Hallo Welt\n")) {}
}
```

22.14.2 Block

Ein Block ist eine Gruppe von möglichen Deklarationen und Anweisungen. Bei *jedem*Eintritt in einen Block werden die Objekte der Deklarationen neu gebildet. Davon sind lediglich Felder mit einer variablen Länge (neu in C99) ausgenommen. Initialisiert werden die Objekte mit der Speicherdauer *automatic storage duration*und Felder mit variabler Länge jeweils zu dem Zeitpunkt, wenn die Programmausführung zu der entsprechenden Deklaration kommt und innerhalb der Deklaration selbst nach den Regeln für Deklarationen der Sprache C. Die Deklaration wird dann wie eine Anweisung betrachtet. Die Speicherdauer der Objekte ist *automatic storage duration*, wenn nicht der Speicherklassenspezifizierer `static`angegeben wurde.

Nach *5.2.4.1 Translation limits*aus C99 wird eine Verschachtelungstiefe bei Blöcken von mindestens 127 Ebenen garantiert.

22.14.3 Siehe auch:

- Kapitel *6.2.4 Storage durations of objects*(Abs. 5) in C99.
- Kontrollstrukturen[177]

22.15 ASCII-Tabelle

Die ASCII-Tabelle enthält alle Kodierungen des ASCII-Zeichensatzes; siehe Steuerzeichen für die Bedeutung der Abkürzungen in der rechten Spalte:

177 Kapitel 6 auf Seite 49

Dez	Hex	Okt	
0	0x00	000	NUL
1	0x01	001	SOH
2	0x02	002	STX
3	0x03	003	ETX
4	0x04	004	EOT
5	0x05	005	ENQ
6	0x06	006	ACK
7	0x07	007	BEL
8	0x08	010	BS
9	0x09	011	TAB
10	0x0A	012	LF
11	0x0B	013	VT
12	0x0C	014	FF
13	0x0D	015	CR
14	0x0E	016	SO
15	0x0F	017	SI
16	0x10	020	DLE
17	0x11	021	DC1
18	0x12	022	DC2
19	0x13	023	DC3
20	0x14	024	DC4
21	0x15	025	NAK
22	0x16	026	SYN
23	0x17	027	ETB
24	0x18	030	CAN
25	0x19	031	EM
26	0x1A	032	SUB
27	0x1B	033	ESC
28	0x1C	034	FS
29	0x1D	035	GS
30	0x1E	036	RS
31	0x1F	037	US

Dez	Hex	Okt	
32	0x20	040	SP
33	0x21	041	!
34	0x22	042	"
35	0x23	043	#
36	0x24	044	$
37	0x25	045	%
38	0x26	046	&
39	0x27	047	'
40	0x28	050	(
41	0x29	051	)
42	0x2A	052	*
43	0x2B	053	+
44	0x2C	054	,
45	0x2D	055	-
46	0x2E	056	.
47	0x2F	057	/
48	0x30	060	0
49	0x31	061	1
50	0x32	062	2
51	0x33	063	3
52	0x34	064	4
53	0x35	065	5
54	0x36	066	6
55	0x37	067	7
56	0x38	070	8
57	0x39	071	9
58	0x3A	072	:
59	0x3B	073	;
60	0x3C	074	<
61	0x3D	075	=
62	0x3E	076	>
63	0x3F	077	?

Dez	Hex	Okt	
64	0x40	100	@
65	0x41	101	A
66	0x42	102	B
67	0x43	103	C
68	0x44	104	D
69	0x45	105	E
70	0x46	106	F
71	0x47	107	G
72	0x48	110	H
73	0x49	111	I
74	0x4A	112	J
75	0x4B	113	K
76	0x4C	114	L
77	0x4D	115	M
78	0x4E	116	N
79	0x4F	117	O
80	0x50	120	P
81	0x51	121	Q
82	0x52	122	R
83	0x53	123	S
84	0x54	124	T
85	0x55	125	U
86	0x56	126	V
87	0x57	127	W
88	0x58	130	X
89	0x59	131	Y
90	0x5A	132	Z
91	0x5B	133	[
92	0x5C	134	\
93	0x5D	135	]
94	0x5E	136	^
95	0x5F	137	_

Dez	Hex	Okt	
96	0x60	140	`
97	0x61	141	a
98	0x62	142	b
99	0x63	143	c
100	0x64	144	d
101	0x65	145	e
102	0x66	146	f
103	0x67	147	g
104	0x68	150	h
105	0x69	151	i
106	0x6A	152	j
107	0x6B	153	k
108	0x6C	154	l
109	0x6D	155	m
110	0x6E	156	n
111	0x6F	157	o
112	0x70	160	p
113	0x71	161	q
114	0x72	162	r
115	0x73	163	s
116	0x74	164	t
117	0x75	165	u
118	0x76	166	v
119	0x77	167	w
120	0x78	170	x
121	0x79	171	y
122	0x7A	172	z
123	0x7B	173	{
124	0x7C	174	|
125	0x7D	175	}
126	0x7E	176	~
127	0x7F	177	DEL

22.16 Literatur

22.16.1 Deutsch:

- **Programmieren in C**Die deutschsprachige Übersetzung des englischen Originals *The C Programming Language*von Brian W. Kernighan[178] und dem C-"Erfinder" Dennis Ritchie[179]. Nach eigener Aussage der Autoren ist das Buch "keine Einführung in das Programmieren; wir gehen davon aus, dass dem Leser einfache Programmierkonzepte - wie Variablen, Zuweisungen, Schleifen und Funktionen - geläufig sind". Der C99-Standard wird nicht berücksichtigt. ISBN 3-446-15497-3

22.16.2 Englisch:

- **The C Programming Language**Das englische Original von Programmierung in C von Brian W Kernighan und Dennis Ritchie. ISBN 0-13-110362-8 (paperback), ISBN 0-13-110370-9 (hardback)
- **The C Standard : Incorporating Technical Corrigendum 1**Das Buch erhält den aktuellen ISO/IEC 9899:1999:TC1 (C99) Standard in gedruckter Form sowie die Rationale. ISBN 0470845732

22.17 Weblinks

22.17.1 Deutsch:

Hilfen beim Einstieg:

- C von A bis Z[180] Openbook von Jürgen Wolf (inklusive Forum zur C- und Linux-Programmierung)
- C-Kurs Interaktiv[181]
- Eine Einführung in C[182]
- C-Tutorial[183]
- Aktiv programmieren lernen mit C[184] von der Freien Universität Berlin
- Programmieren in C - Eine Einführung[185] Eine stichwortartige Einführung in C von Peter Klingebiel
- C und C++ für UNIX, DOS und MS-Windows (3.1, 95, 98, NT)[186] von Prof. Dr. Dankert

Webseiten zum Nachschlagen:

178 http://de.wikipedia.org/wiki/Brian%20W.%20Kernighan%20
179 http://de.wikipedia.org/wiki/Dennis%20Ritchie%20
180 http://www.pronix.de/modules/C/openbook/
181 http://www.fh-fulda.de/~klingebiel/c-kurs/
182 http://www.stud.tu-ilmenau.de/~schwan/cc/node1.html
183 http://info.baeumle.com/ansic.html
184 http://www.purl.org/stefan_ram/pub/c_de
185 http://www.fh-fulda.de/~klingebiel/c-vorlesung/index.htm
186 http://www.haw-hamburg.de/rzbt/dankert/c_tutor.html/

- Übersicht über den C99-Standard[187]
- ANSI-C im Überblick[188] von Peter Baeumle-Courth

FAQs:

- FAQ der deutschsprachigen Newsgroup de.comp.lang.c[189] (berücksichtigt nicht den C99-Standard)

Und abschließend noch etwas zum Lachen für geplagte C-Programmierer:

- Erfinder von UNIX und C geben zu: ALLES QUATSCH![190] Aber Vorsicht: Satire!

22.17.2 Englisch:

Hilfen beim Einstieg:

- The C Book[191] von Mike Banahan, Declan Brady und Mark Doran
- Howstuffworks/C[192] Kleines online Tutorial mit anschaulichen Beispielen

C-Standardbibliothek:

- Dinkum C99 Library Reference Manual[193]
- Die C-Standard-Bibliothek[194]
- The C Library Reference Guide[195] von Eric Huss
- Dokumentation der GNU C Library[196]

Entstehung von C:

- *Programming in C*eine der frühesten Versionen[197]
- Homepage von Dennis Ritchie[198]
  - Die Geschichte der Sprache C[199]
  - Martin Richards's BCPL Reference Manual, 1967 Martin Richards's BCPL Reference Manual, 1967[200]

C99 Standard:

- The New C Standard - An Economic and Cultural Commentary[201] Sehr ausführliche Beschreibung des C-Standards (ohne Bibliotheksfunktionen) von Derek M. Jones (PDF)

187 http://www.schellong.de/c.htm
188 http://www.petra-budde.de/download/ansi-c.pdf
189 http://www2.informatik.uni-wuerzburg.de/dclc-faq/
190 http://home.nikocity.de/schmengler/presse/quatsch.htm
191 http://publications.gbdirect.co.uk/c_book/
192 http://www.howstuffworks.com/c.html
193 http://www.dinkumware.com/c99.aspx
194 http://www.infosys.utas.edu.au/info/documentation/C/CStdLib.html#Contents
195 http://www.acm.uiuc.edu/webmonkeys/book/c_guide/index.html
196 http://www.gnu.org/software/libc/manual/
197 http://www.lysator.liu.se/c/bwk-tutor.html
198 http://www.cs.bell-labs.com/who/dmr/index.html
199 http://cm.bell-labs.com/cm/cs/who/dmr/chist.html
200 http://cm.bell-labs.com/cm/cs/who/dmr/bcpl.html
201 http://homepage.ntlworld.com/dmjones/cbook1_0a.pdf

- Are you ready for C99?[202] Die Neuerungen des C99 Standards im Überblick
- Open source development using C99[203] Sehr ausführlicher Überblick über die Neuerungen des C99-Standards von Peter Seebach
- Incompatibilities Between ISO C and ISO C++[204] von David R. Tribble

Verschiedenes:

- Sequence Points[205] Artikel von Dan Saks
- How to Write Unmaintainable Code[206] Satirische Anmerkungen zum Programmierstil, insb. zur Namensgebung von Symbolen, von Roedy Green

22.18 Newsgroup

Bei speziellen Fragen zu C bekommt man am Besten über eine Newsgroup qualifizierte Hilfe. Bitte beachten Sie, dass es auf den Newsgroups de.comp.lang.c und comp.lang.c nur um ANSI C geht. Systemabhängige Fragen werden äußerst ungern gesehen und werden in der Regel gar nicht erst beantwortet. Bevor Sie posten, lesen Sie sich bitte erst die FAQ der Newsgroups durch (siehe Weblinks). Bei Fragen zu ANSI C hilft man aber gerne weiter.

Deutsch:

- news:de.comp.lang.c

Englisch:

- news:comp.lang.c

22.19 Der C-Standard

Der C-Standard ist nicht frei im Netz verfügbar. Man findet im WWW zwar immer wieder eine Version des ISO/IEC 9899:1999 (C99)-Standards, hierbei handelt es sich in der Regel allerdings nur um einen Draft, der in einigen Punkten nicht mit dem tatsächlichen Standard übereinstimmt. Der Standard kann nur kostenpflichtig über das ANSI-Institut bezogen werden. Dessen Webadresse lautet:

- `http://www.ansi.org/`

Dort kann man ihn unter der folgenden URL beziehen:

- `http://webstore.ansi.org/ansidocstore/product.asp?sku=ISO%2FIEC+9899%3A1999`

Als wesentlich günstiger erweist sich hier die gedruckte Variante (siehe Literatur[207]).

202 `http://www.kuro5hin.org/story/2001/2/23/194544/139`
203 `http://www-106.ibm.com/developerworks/linux/library/l-c99.html?ca=dgr-lnxw07UsingC99`
204 `http://david.tribble.com/text/cdiffs.htm`
205 `http://www.embedded.com/shared/printableArticle.jhtml?articleID=9900661`
206 `http://thc.org/root/phun/unmaintain.html`
207 Kapitel 22.17.2 auf Seite 215

Da auch der Standard nicht perfekt ist, werden in unregelmäßigen Abständen die Fehler verbessert und veröffentlicht. Solche Überarbeitungen werden als Technical Corrigendum (kurz TC) bezeichnet, und sind vergleichbar mit einem Errata. Der TC für den ISO/IEC 9899:1999-Standard ist unter der folgenden Webadresse frei verfügbar:

- http://www.open-std.org/jtc1/sc22/wg14/www/docs/9899tc1/n32071.PDF[208]

Neben dem eigentlichen Standard veröffentlicht das ANSI-Komitee sogenannte Rationale. Dabei handelt es sich um Erläuterungen zum Standard, die das Verständnis des recht schwer lesbaren Standards erleichtern soll und Erklärungen erhält warum etwas vom Komitee beschlossen wurden. Sie sind nicht Teil des Standards und deshalb frei im Web verfügbar. Unter der folgenden Webadresse können die Rationale zum C99-Standard bezogen werden:

- http://www.open-std.org/jtc1/sc22/wg14/www/C99RationaleV5.10.pdf[209]

Obwohl der originale Standardtext nicht frei verfügbar ist, wurde von der C-Standard-Arbeitsgruppe (WG14) mittlerweile eine Version auf deren Webseite bereitgestellt, die laut ihrer eigenen Aussage dem verabschiedeten Standard einschließlich der beiden Überarbeitungen entspricht. Diese ist unter der folgenden Webadresse verfügbar:

- http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1124.pdf[210]

Hinweis: Weitere Fragen bitte einfach in eine Kategorie oder ganz unten einfügen. Es müssen dabei keine Antworten mit angegeben werden.

22.20 Fragen zu diesem Buch

22.20.1 Ich habe in einem anderen Buch gelesen, dass ...

In diesem Fall solltest du hingehen und den Abschnitt verbessern. Allerdings gibt es eine ganze Reihe sehr populärer Irrtümer über C und du solltest deshalb vorher anhand des Standards überprüfen, ob die Aussage tatsächlich zutrifft. Hier nur ein unvollständige Liste der populärsten Irrtümer:

- Ein Programm beginnt mit `void main(void)`, `main()`usw. – Dies entspricht nicht dem (C99)-Standard. Dort ist festgelegt, dass jedes Programm (sofern ihm keine Parameter übergeben werde) mit `int main()`oder `int main(void)`beginnen **muss**. Die Definition mit `void main()`bzw. `void main(void)`ist kein gültiges C, da der Standard vorschreibt, dass `main`einen Rückgabewert vom Typ `int`besitzen muss (auch wenn viele Compiler `void`dennoch akzeptieren). Die Definition mit `main()`war früher gültig, da beim Fehlen eines Rückgabetyps angenommen wurde, dass die Funktion `int`zurückliefert.
- Jeder C-Compiler besitzt eine Headerdatei mit dem Namen `stdio.h`. – Dies ist falsch. Der Standard sagt ganz klar: *A header is not necessarily a source file, nor are the <and >delimited sequence in header names necessarily valid source file names.*(Abschnitt 7.1.2 Standard header Fußnote 154). Es muss also keine Datei mit dem Namen `stdio.h`geben. Das Selbe trifft natürlich auch auf die anderen Headerdateien zu.

208 `http://www.open-std.org/jtc1/sc22/wg14/www/docs/9899tc1/n32071.PDF`
209 `http://www.open-std.org/jtc1/sc22/wg14/www/C99RationaleV5.10.pdf`
210 `http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1124.pdf`

- Der Variablentyp xyz hat die Größe von xyz Byte. – Auch dies ist falsch. Der Standard legt lediglich fest, dass `char`1 Byte groß ist. Für die anderen Typen sind lediglich Mindestgrößen festgelegt. Erst mit dem C99-Standard wurden Variablen wie beispielsweise `int8_t`oder `int16_t`eingeführt, die eine feste Größe besitzen.
- Eine Variable des Typs `char`ist 8 Bit breit. – Auch dies ist genaugenommen nicht richtig. Einige Autoren behaupten dann noch, dass ein Byte in C eine beliebige Zahl von Bits haben kann. Richtig dagegen ist, dass in C ein Byte mindestens aus 8 Bit bestehen muss. Tatsächlich kann man dies aber häufig vernachlässigen; K&R erwähnen dies in ihrem Buch auch nicht gesondert. Wer dennoch hochportable Programme schreiben möchte und die Anzahl der Bits benötigt, kann dies wie folgt ermitteln:

```
#include <limits.h> // für CHAR_BIT
size=sizeof(datentyp) * CHAR_BIT;
```

22.21 Variablen und Konstanten

22.21.1 Es heißt, dass der Ausdruck `sizeof(char)`immer den Wert 1 liefert, also der Typ `char`immer die Größe von 1 Byte hat. Dies ist aber unlogisch, da UNICODE Zeichen 16 Bit und damit 2 Byte besitzen. Hier widerspricht sich der Standard doch, oder?

Nein, tut er nicht. Der Denkfehler liegt darin anzunehmen, dass ein UNICODE Zeichen in einem `char`abgelegt werden muss. In der Regel wird es aber in einem `wchar_t`abgelegt. Dies ist laut C - Standard ein ganzzahliger Typ, dessen Wertebereich ausreicht, die Zeichen des größten erweiterten Zeichensatzes der Plattform aufzunehmen. Per Definition liefert `sizeof(char)`**immer**den Wert 1.

22.21.2 Welche Größe hat der Typ `int`auf einem 64 Bit Prozessor ?

Der Standard legt die Größe des Typs `int`nicht fest. Er sagt nur: *A "plain"* ***int****object has the natural size suggested by the architecture of the execution environment*(Abschnitt 6.2.5 Types Absatz 4). Man könnte daraus schließen, dass `int`auf einer 64 Bit Plattform 64 Bit groß ist, was aber nicht immer der Fall ist.

22.21.3 Es ist mir immer noch nicht ganz klar, was dieses EOF Zeichen bedeutet.

EOF ist ein negativer Integerwert, der von einigen Funktionen geliefert wird, wenn das Ende eines Stroms erreicht worden ist. Bei Funktionen, mit denen auf Dateien zugegriffen werden kann, ist EOF das End of File – also das Dateiende.

Der Denkfehler einiger Anfänger liegt vermutlich darin, dass EOF Zeichen grundsätzlich mit dem Dateiende gleichzusetzen. EOF kennzeichnet aber ganz allgemein das Ende eines Stroms, zu dem auch der Eingabestrom aus der Standardeingabe (Tastatur) gehört. Deshalb kann auch `getchar`EOF liefern.

22.22 Operatoren

22.22.1 Mir ist immer noch nicht ganz klar, warum `a = i + i++`ein undefiniertes Resultat liefert. Der `++`- Operator hat doch eine höhere Priorität als der `+`-Operator.

Ja, es ist richtig, dass der `++`Operator eine höhere Priorität als der `+`-Operator hat. Der Compiler errechnet deshalb zunächst das Ergebnis von `i++`. Allerdings müssen die Nebenwirkungen erst bis zum Ende des Ausdrucks ausgewertet worden sein.

Anders ausgedrückt: Der C-Standard schreibt dem Compiler nicht vor, wann er den Wert von `i`ändern muss. Dies kann sofort nach der Berechnung von `i++`sein oder erst am Ende des Ausdrucks beim Erreichen des Sequenzpunktes.

Dagegen hat

```
b = c = 3;
a = b + c++;
```

ein klar definiertes Ergebnis (nämlich 6), da die Variable `c`nicht an einer anderen Stelle vor dem Sequenzpunkt verändert wird.

Es ist etwas anders: Das Inkrement kennt zwei unterschiedliche Anwendungen, nämlich before (++i) und after (i++) bei der Durchführung, daher hat

```
b = c = 3;
a = b + ++c;
```

auch ein klar definiertes Ergebnis (nämlich 7)!

22.23 Zeiger

22.23.1 Ist bei `malloc`ein Cast unbedingt notwendig? Ich habe schon öfter die Variante `zeiger = (int*) malloc(sizeof(int) * 10);`genauso wie `zeiger = malloc(sizeof(int) * 10);`gesehen.

Für diese Antwort muss man etwas ausholen: In der ersten Beschreibung der Sprache von K&R gab es noch keinen Zeiger auf `void`. Deshalb gab `malloc`einen `char*`zurück und ein cast auf andere Typen war notwendig. Es konnte deshalb nur die erste Variante `zeiger = (int*) malloc(sizeof(int) * 10);`eingesetzt werden.

Mit der Standardisierung von C wurde der untypisierte Zeiger `void*`eingeführt, der in jeden Typ gecastet werden kann. Daher ist kein expliziter Cast mehr notwendig und es kann die zweite Variante `zeiger = malloc(sizeof(int) * 10);`benutzt werden. K&R benutzen in ihrem Buch allerdings auch in der aktuellen Auflage die erste der beiden Varianten und behauptet, dass dieser Typ explizit in den gewünschten Typ umgewandelt werden muss. Dies ist aber einer der wenigen Fehler des Buches und wird vom ANSI C Standard nicht gefordert. Leider wird diese falsche Behauptung oft von vielen Büchern übernommen.

Es gibt allerdings dennoch einen Grund `void*`zu casten, und zwar dann wenn ein C++-Compiler zum Übersetzen benutzt werden soll. Da wir uns in diesem Buch allerdings an ANSI C halten, benutzen wir keinen Cast.

Dieses Wikibooks besitzt keinen eigenen Glossar. Stattdessen wird auf Artikel in der freien Internetenzyklopädie Wikipedia verwiesen. Die Einträge sind dort umfangreicher, als wenn speziell für dieses Buch ein Glossar erstellt worden wäre.

- Algorithmus [211]
- ANSI C[212]
- BCPL[213]
- Binder[214]
- Brian W. Kernighan[215]
- C[216]
- C++[217]
- C99[218]
- Compiler[219]
- Computerprogramm[220]
- Datentyp[221]
- Debugger[222]
- Deklaration [223]
- Dennis Ritchie[224]
- Endlosschleife[225]
- For-Schleife[226]
- Funktion[227]
- GNU Compiler Collection[228]
- Goto[229]
- Hallo-Welt-Programm[230]
- Integer[231]

211 http://de.wikipedia.org/wiki/Algorithmus%20
212 http://de.wikipedia.org/wiki/ANSI%20C%20%20
213 http://de.wikipedia.org/wiki/BCPL%20
214 http://de.wikipedia.org/wiki/Linker%20%28Computerprogramm%29%20
215 http://de.wikipedia.org/wiki/Brian%20W.%20Kernighan%20
216 http://de.wikipedia.org/wiki/C%20%28Programmiersprache%29%20
217 http://de.wikipedia.org/wiki/C-Plusplus%20
218 http://de.wikipedia.org/wiki/ISO%2FIEC%209899%3A1999%20
219 http://de.wikipedia.org/wiki/Compiler%20
220 http://de.wikipedia.org/wiki/Computerprogramm%20
221 http://de.wikipedia.org/wiki/Datentyp%20
222 http://de.wikipedia.org/wiki/Debugging%20
223 http://de.wikipedia.org/wiki/Deklaration%20%28Programmierung%29%20
224 http://de.wikipedia.org/wiki/Dennis%20Ritchie%20
225 http://de.wikipedia.org/wiki/Endlosschleife%20
226 http://de.wikipedia.org/wiki/For-Schleife%20
227 http://de.wikipedia.org/wiki/Funktion%20%28Programmierung%29%20
228 http://de.wikipedia.org/wiki/GNU%20Compiler%20Collection%20
229 http://de.wikipedia.org/wiki/Goto%20
230 http://de.wikipedia.org/wiki/Hallo-Welt-Programm
231 http://de.wikipedia.org/wiki/Integer%20%28Datentyp%29%20

- Ken Thompson[232]
- Kontrollstrukturen[233]
- Linker[234]
- Makro[235]
- Portierung[236]
- Präprozessor[237]
- Programmierfehler[238]
- Programm[239]
- Programmiersprache[240]
- Quelltext[241]
- Schleife[242]
- Spaghetticode[243]
- Standard C Library[244]
- Strukturierte Programmierung[245]
- Syntax[246]
- Unix[247]
- Variable[248]
- Verzweigung[249]
- While-Schleife[250]
- Zeichenkette[251]
- Zeiger[252]

232 http://de.wikipedia.org/wiki/Ken%20Thompson%20
233 http://de.wikipedia.org/wiki/Kontrollstrukturen%20
234 http://de.wikipedia.org/wiki/Linker%20%28Computerprogramm%29%20
235 http://de.wikipedia.org/wiki/Makro%20
236 http://de.wikipedia.org/wiki/Portierung%20
237 http://de.wikipedia.org/wiki/Pr%C3%A4prozessor%20
238 http://de.wikipedia.org/wiki/Programmfehler%20
239 http://de.wikipedia.org/wiki/Computerprogramm%20
240 http://de.wikipedia.org/wiki/Programmiersprache%20
241 http://de.wikipedia.org/wiki/Quelltext%20
242 http://de.wikipedia.org/wiki/Schleife%20%28Programmierung%29%20
243 http://de.wikipedia.org/wiki/Spaghetticode%20
244 http://de.wikipedia.org/wiki/Standard%20C%20Library%20
245 http://de.wikipedia.org/wiki/Strukturierte%20Programmiersprache%20
246 http://de.wikipedia.org/wiki/Syntax%20
247 http://de.wikipedia.org/wiki/Unix%20
248 http://de.wikipedia.org/wiki/Variable%20%28Programmierung%29%20
249 http://de.wikipedia.org/wiki/Verzweigung%20%28Programmierung%29%20
250 http://de.wikipedia.org/wiki/While-Schleife%20
251 http://de.wikipedia.org/wiki/Zeichenkette%20
252 http://de.wikipedia.org/wiki/Zeiger%20%28Informatik%29%20

23 Aufgaben

23.1 Sinuswerte

23.1.1 Aufgabenstellung

Entwickeln Sie ein Programm, das Ihnen die Werte der Sinusfunktion in 10er Schritten von 0 bis 360° mit drei Stellen nach dem Komma ausgibt. Die Sinusfunktion `sin()`ist in der Header-Datei math.h definiert. Achten Sie auf eventuelle Typkonvertierungen.

23.1.2 Musterlösung

```
#include <stdio.h>
#include <math.h>
#define PI 3.14159f // Konstante PI

int main(void)
{
    // Variablen deklarieren
    float winkel;
    float rad;
    float sinus;

    printf("Programm zur Berechnung der Sinusfunktion in 10er Schritten\n");
    printf("Winkel \t\t Sinus des Winkel\n");

    // Schleife zur Berechnung der Sinuswerte
    int i;
    for (i = 0; i <= 36; i++)
    {
        winkel = 10 * i; // 10er Schritte berechnen

        rad = winkel * PI / 180; // Berechnen des Bogenmaßwinkels
        sinus = sin(rad); // Ermitteln des Sinuswertes

        printf("%g \t\t %.3f\n", winkel, sinus); // tabellarische Ausgabe
    }

    return 0;
}
```

Wir benutzen bei der Musterlösung drei Variablen:

1. *winkel*für die Berechnung der Winkel in 10er Schritten,
2. *rad*zur Berechnung des Bogenmaßes und
3. *sinus*für den endgültigen Sinuswert.

In einer Schleife werden die Winkel und deren Sinuswerte nacheinander berechnet. Anschließend werden die Winkel tabellarisch ausgegeben.

23.2 Dreieick

23.2.1 Aufgabenstellung

Entwickeln Sie ein Programm, das ein auf der Spitze stehendes Dreieck mit Sternchen (*) auf dem Bildschirm in folgender Form ausgibt:

```
*******
 *****
  ***
   *
```

Durch eine manuelle Eingabe zu Beginn des Programmes muss festgelegt werden, aus wie vielen Zeilen das Dreieck aufgebaut werden soll. Anschließend muss überprüft werden, ob die Eingabe gültig ist. Ist das nicht der Fall, muss das Programm abgebrochen werden.

Zur Implementierung der Aufgabe werden neben Ein- und Ausgabe auch Schleifen benötigt.

23.2.2 Musterlösung

```
#include <stdio.h>

int main(void)
{
    // Deklarationen
    int i, j, k;
    int hoehe; // Variable fuer die Dreieckshoehe
    int anzahlSterne, anzahlLeer; // Variablen zur Speicherung von Sternen und
 Leerzeichen


    // Eingabe der Dreieckshoehe
    printf("Programm zur Ausgabe eines auf der Spitze stehendes Dreiecks\n");
    printf("Bitte die Hoehe des Dreiecks eingeben: ");
    scanf("%d", &hoehe); // Eingabeaufforderung für Dreieckshoehe

    if ((!hoehe) || (hoehe <= 0)) // Ist Eingabe gueltig?
    {
        printf("Ungueltige Eingabe!\n");
        return 1;
    }

    // Schleife zur Ausgabe
    for (i = 1; i <= hoehe; i++) // Hauptschleife zum Aufbau des Dreiecks
    {
        // Fuer jede neue Zeile die Anzahl der notwendigen Sterne und
 Leerzeichen ermitteln
        anzahlLeer = i;
        anzahlSterne = (hoehe + 1 - i) * 2 - 1;

        for (j = 1; j <= anzahlLeer; j++) // Ausgabe der Leerzeichen
            printf(" ");

        for (k = 1; k <= anzahlSterne; k++)        // Ausgabe der Sterne
            printf("*");

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

```
    return 0;
}
```

Wir deklarieren zu Beginn drei Variablen:

1. *hoehe*für die Anzahl der Zeilen über die sich das Dreieck erstreckt
2. *anzahlSterne*für die Anzahl der Sterne in jeder Zeile
3. *anzahlLeer*für die Anzahl der Leerzeichen in jeder Zeile.

Als Nächstes benötigen wir die Eingabe der Dreieckshöhe. Dazu wird über ein `scanf()`eine Zahl eingelesen und in der Variable *hoehe*gespeichert. Anschließend wird die Eingabe mit `if`überprüft. Wenn *hoehe*leer ist, weil die Eingabe keine Zahl war, oder die Zahl kleiner gleich Null ist, wird ein Fehler ausgegeben und das Programm beendet.

Ist die Eingabe gültig, kommen wir zur Hauptschleife (`for`). Diese wird für jede Zeile einmal abgearbeitet. Hier wird nun für jede Zeile die Anzahl der benötigten Sterne und Leerzeichen ermittelt. Jede Zeile beginnt mit Leerzeichen, weshalb diese zuerst mit einer `for`-Schleife ausgegeben werden. Darauf folgt eine weitere `for`-Schleife, welche die Anzahl der Sterne ausgibt. Am Ende der Hauptschleife erfolgt ein Zeilenumbruch.

Ist die Hauptschleife durchlaufen, wird das Programm erfolgreich beendet.

23.3 Vektoren

23.3.1 Aufgabenstellung

Entwickeln Sie ein Programm, das das Skalarprodukt zweier Vektoren bestimmt. Die Anzahl der Elemente und die Werte der Vektoren sind in der Eingabeschleife manuell einzugeben.

Überprüfen Sie, ob die Anzahl der Elemente die Maximalgröße der Vektoren überschreitet und ermöglichen Sie ggf. eine Korrektur. Legen Sie die maximale Anzahl der Vektorelemente mit einer define-Anweisung durch den Präprozessor fest.

Skalarprodukt: $A*B = a_1*b_1 + a_2*b_2 + a_3*b_3 + ... + a_n*b_n$

23.3.2 Musterlösung

```
#include <stdio.h>
#define DIMENSION 100
int main (void)
{
    int v1[DIMENSION],v2[DIMENSION];  // Arrays für Vektor 1 und 2
    int anzahl;      // Dimension der Vektoren
    int index;       //Zählwert der Arrays;
    int produkt;     //Produkt jedes Schleifendurchlaufs;
    int ergebnis=0;  //Gesamtwert auf den Einzelprodukte aufaddiert werden

    // Programmüberschrift
    printf("Skalarprodukt 2er beliebiger Vektoren berechnen\n\n");

    do
    {
        printf("Bitte Anzahl der Dimensionen angeben (1-%i):" ,DIMENSION);
        scanf("%i",&anzahl);  //Einlesen des Wertes der Vektordimension
```

```
        if (anzahl>DIMENSION || anzahl<1)  //wenn Wert > DIMENSION erneute
 Eingabe
            {
                printf("\n Eingabe uebersteigt max. Dimensionszahl\n\n");

            }

    }while (anzahl>DIMENSION || anzahl<1);

    for(index=0; index<anzahl; index++)  //Einleseschleife des ersten Vektors
        {
            printf("Wert %i fuer vektor 1 eingeben: ",index+1);
            scanf("%i",&v1[index]);  //Einlesen des Vektorwertes
        }

    for(index=0; index<anzahl; index++)  // Einleseschleife des zweiten Vektors
        {
            printf("Wert %i fuer Vektor 2 eingeben: ",index+1);
            scanf("%i",&v2[index]);  //Einlesen eines Vektorwertes
        }

    //Schleife zur Berechnung des Skalarproduktes
    for(index=0; index<anzahl; index++)
        {
            produkt=v1[index]*v2[index];  //Einzelwerte addieren
            ergebnis+=produkt;  //Produkte zum Gesamtwert aufsummieren
        }
/*Berechnung kann auch in die letzte Eingabeschleife integriert werden*/
    // Ausgabe des Gesamtwertes des Skalarproduktes
    printf("Das Skalarprodukt der Vektoren betraegt: %i\n",ergebnis);

return 0;
}
```

23.4 Polygone

23.4.1 Augabenstellung

Geometrische Linien können stückweise gerade durch Polygonzüge approximiert werden. Eine Linie kann dann so durch eine Menge von Punkten beschrieben, die die Koordinaten der End- und Anfangspunkte der geradem Abschnitte darstellen. Die Punkte eines Polygonzuges sind in einem Array gespeichert, das die maximale Anzahl von N Elementen hat. N soll als symbolische Konstante verwendet werden. Jeder Punkt soll durch eine Strukturvariable, die die x- und y-Koordinaten als Komponenten hat, beschrieben werden. Eine Linie wird also durch einen Vektor, dessen Elemente Strukturen sind, beschrieben.

Entwickeln Sie ein Programm, dass folgende Funktionen beinhaltet.

- Manuelle Eingabe der Punktkoordinaten eines Polygons.
- Bestimmung der Länge des Polygons und Ausgabe des Wertes auf dem Bildschirm.
- Tabellarische Ausgabe der Punktkoordinaten eines Polygons auf dem Bildschrim.

Die Auswahl der Funktionen soll durch ein Menü erfolgen. Verwenden Sie dazu die switch-Konstruktion.

23.4.2 Musterlösung

```
#include <stdio.h>
#include <math.h>
#define PUNKTE 1000  //Definieren einer Konstanten



typedef struct koordinate  //Definieren der Structur "POLYGON"
{
    int  x; //kordinate x
    int  y; //kordinate y
} POLYGON;

//Deklarieren der Unterfunktionen
int einlesen( POLYGON p[PUNKTE] );
void ausgabe (int anzahlpunkte,POLYGON p[PUNKTE]);
float berechnung (int anzahlpunkte, POLYGON p[PUNKTE] );


/* Beginn der Hauptfunktion*/
int main (void)

{

    POLYGON p[PUNKTE];
    int anzahlpunkte;
    int menuezahl;


    printf("Dies ist ein Programm zur Berechnung eines Polygonzuges\n\n");
    do
    {
        /* Eingabe menue*/
        printf("*********************************************************\n");
        printf("*  Sie haben folgende Moeglichkeiten:\t\t\t*\n");
        printf("*  1: Eingabe von Werten zur Berechnung des Polygones\t*\n");
        printf("*  2: Ausgabe der eingegebenen Werte in Tabellenform\t*\n");
        printf("*  3: Berechnen des Polygonzuges\t\t\t*\n");
        printf("*  4: Beenden des Programmes\t\t\t\t*\n");
        printf("*  Bitte geben sie eine Zahl ein!\t\t\t*\n");
        printf("*********************************************************\n");
        scanf("%d",&menuezahl);

        switch(menuezahl)
        {
        case 1:        //Funktionsaufruf: Einlesen der Punktkoordinaten
            anzahlpunkte = einlesen( p );
            break;

        case 2:        //Funktionsaufruf: Ausgabe der eingelesenen Werte
            ausgabe(anzahlpunkte,p);
            break;

        case 3:        //Funktionsaufruf: Berechnung des Polygonzuges

            printf("der eingegebene Polygonzug ist %f lang.\n\n",berechnung
 (anzahlpunkte,p));
            break;

        case 4:         //Beenden der Funktion
            printf("Auf Wiedersehen, benutzen sie dieses Programm bald
 wieder!\n\n");
            break;

        default:  // bei falscher Eingabe
```

```
            printf("Ihrer Eingabe konnte kein Menüpunkt zugeordnet
 werden!\nBitte versuchen sie es erneut.\n");

        }

    }while(menuezahl!=4); //Ende der Schleife bei Eingabe der Zahl 4

    return 0;
}


int einlesen( POLYGON p[PUNKTE] ) //Funktion: Einlesen der Koordinaten
{
    int zeile;
    int anzahlpunkte;

    do
    {
        printf("Bitte geben sie die Anzahl der Punkte des Polygons ein.\nBitte
 beachten sie, dass es min. 2 Punkte aber max. %i Punkte sein müssen!",PUNKTE);
        scanf("%i",&anzahlpunkte);

        // Entscheidung, ob eingegebene Zahl verarbeitet werden kann
        if (anzahlpunkte<2 || anzahlpunkte>PUNKTE)
            printf("falsche eingabe!\n\n");

    }while(anzahlpunkte<2 || anzahlpunkte>PUNKTE);


    for (zeile=0;zeile<anzahlpunkte;zeile++)  //Koordinaten für Berechnung
 einlesen
    {
        printf(" wert %d fuer x eingeben:",zeile+1);
        scanf("%d",&p[zeile].x);
        printf( " wert %d fuer y eingeben:",zeile+1);
        scanf("%d",&p[zeile].y);
    }
    printf("\n");
    return anzahlpunkte;
}


// Funktion zur Ausgabe der eingelesenen Punkte
void ausgabe (int anzahlpunkte,POLYGON p[PUNKTE] )
{
    int zeile;

    printf("Anzahl\t|   x werte \t|   y werte\n");
    //Schleife zum Auslesen der Struktur und Ausgabe der Tabelle
    for (zeile=0;zeile<anzahlpunkte;zeile++)
    {
        printf(" %5d\t|\t",zeile+1);
        printf(" %5d\t|\t",p[zeile].x);
        printf(" %5d\n",p[zeile].y);
    }
    printf("\n");

}


//Funktion zum Berechnen des Polygons aus den eingelesenen Werten
float berechnung (int anzahlpunkte, POLYGON p[PUNKTE])
{
    float ergebnis;
    int zeile;
    float c;
```

```
    ergebnis=0;
    //Schleife zum Auslesen und Berechnen der Punkte
    for (zeile=0;zeile<anzahlpunkte-1;zeile++)
    {
        c = (float)sqrt(pow(p[zeile].x - p[zeile+1].x,2) + pow(p[zeile+1].y -
 p[zeile].y,2)); //pow(x,y) x^y
        ergebnis+=c;  //Gleichung zum Berechnen des Polygons
    }
    return ergebnis ;
}
```

23.5 Letztes Zeichen finden

23.5.1 Aufgabenstellung

Schreiben Sie eine Funktion, die feststellt, an welcher Stelle einer Zeichenkette ein Buchstabe das letzte Mal vorkommt. Als Parameter für die Funktion soll ein Zeiger auf den Anfang der Zeichenkette und das zu suchende Zeichen übergeben werden. Die Stellennummer, an der das Zeichen das letzte Mal vorkommt, ist der Rückgabewert. Ist das Zeichen nicht vorhanden oder wird ein Nullpointer an die Funktion übergeben, soll der Wert -1 geliefert werden. Testen Sie die Funktion in einem kurzen Hauptprogramm.

23.5.2 Musterlösung

```
#include <stdio.h>
#define LAENGE 1234

int position(char *zeichenkette, char zeichen); // Prototyp der Suchfunktion

int main(void)
{
    int position_zeichen,start,c;  //Deklaration der Variablen
    char zeichen, zeichenkette[LAENGE];
    printf("Das ist ein Programm zum Vergleich einer Zeichenkette mit einem
 Zeichen\n");


    printf("Bitte Zeichenkette mit maximal %d Zeichen eingeben: ",LAENGE-1);
    // Einlesen einer beliebigen Zeichenkette mit Sonderzeichen
    for(start=0;(start<LAENGE-1) && ((c=getchar()) != EOF) &&c!='\n' ;start++)
    {
        zeichenkette[start]=(char)c;

    }
    zeichenkette[start] = '\0'; //Nullbyte an letzter Stelle hinzufügen

    if(start==LAENGE-1) //Wenn zu viele Zeichen sind, hier verarbeiten
    {
        while(getchar()!='\n'); // Zeichen solange einlesen bis Enter
    }

    printf("Bitte ein Zeichen eingeben:");

    scanf("%c",&zeichen);  //Einlesen des gesuchten Zeichens

    position_zeichen = position(zeichenkette,zeichen); //Übergabe des
 Rückgabewertes aus der Funktion 'position'
```

```
    if (position_zeichen == -1)        //ist das Zeichen vorhanden?
        printf("Eingegebenes Zeichen ist nicht in der Zeichenkette
 enthalten!\n");
    else
        // wenn ja, Ausgabe des Suchergebnisses
        printf("Position des letzten %c ist an Stelle: %i\n", zeichen,
 position_zeichen+1);

    position(NULL,zeichen);
    return 0;
}

//Funktion zum Suchen des Zeichens
int position(char *zeichenkette, char zeichen)
{
    int back = -1,i;

    if(zeichenkette!=NULL) //Wenn keine Zeichen vorhanden sind, Rückgabe von 0
    {
        //Schleife zum Durchgehen der Zeichenkette
        for(i = 0; *(zeichenkette+i) != '\0'; i++)
        {
            printf("An Stelle %4d steht das Zeichen  =
 %c\n",i+1,*(zeichenkette+i));//Kontrollausgabe der Zeichen mit der zugewiesenen
 Positionszahl
            // Vergleich der einzelnen Zeichen mit dem gesuchten
            if (*(zeichenkette+i) == zeichen)
            {
                back = i; //Position des gesuchten Zeichens speichern
            }
        }
    }
    return back;        //Rückgabe der Position des gesuchten Zeichens
}
```

23.6 Zeichenketten vergleichen

23.6.1 Aufgabenstellung

Schreiben Sie ein Programm, das zwei eingelesene Zeichenketten miteinander vergleicht. Die Eingabe der Zeichenketten soll durch eine Schleife mit Einzelzeicheneingabe realisiert werden. Als Ergebnis sollen die Zeichenketten in lexikalisch richtiger Reihenfolge ausgegeben werden. Beide Zeichenketten sollen über Zeiger im Hauptspeicher zugänglich sein.

Verwenden Sie für die Eingabe einer Zeichenkette einen statischen Zwischenpuffer. Nach Beendigung der Zeichenketteneingabe in diesen Puffer soll der notwendige Speicherplatz angefordert werden und die Zeichenkette in den bereitgestellten freien Speicherplatz übertragen werden.

Hinweis: Informieren Sie sich über den Gebrauch der Funktionen malloc() und free().

23.6.2 Musterlösung

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

```
#define LAENGE 5

char* einlesen (int j);

int main (void)

{
    char *zeichenkette1_gespeichert=NULL,*zeichenkette2_gespeichert=NULL
    char *temp2,*temp1,temp3,temp4;
    int start,a;


    do {

        printf("In diesem Programm koennen Sie 2 kleingeschriebene
            Zeichenketten mit jeweils\nmaximal %i Zeichen
            lexikalisch sortieren lassen.\n",LAENGE);

        //Einlesen der Zeichenketten
        zeichenkette1_gespeichert=einlesen(1);

        if (zeichenkette1_gespeichert==NULL){
            printf("\n\nEs konnte kein ausreichender Speicher
                zur Verfuegung gestellt werden.
                \nDas Programm wird beendet.\n");
            break;
        }

        zeichenkette2_gespeichert=einlesen(2);

        if (zeichenkette2_gespeichert==NULL){
            printf("\n\nEs konnte kein ausreichender Speicher
            zur Verfuegung gestellt werden.
            \nDas Programm wird beendet.\n");
            break;
        }

        // Sortieren der Zeichenketten lexikalisch
        start=1;
        //Übergeben der Zeichenkette an temp
        temp1=zeichenkette1_gespeichert;
        temp2=zeichenkette2_gespeichert;
        a=0;

        while (*temp1!='\0'&&*temp2!='\0'&&a==0)
        {
            temp3=*(temp2);        //Inhalt Übergabe Variable
            temp4=*(temp1);        //Inhalt Übergabe Variable

            if(temp4>temp3)
                a=1;
            if(temp4<temp3)
                a=2;
            temp1++;        //Adresse von Zeiger um 1 weiterschieben
            temp2++;
        };

        printf("\nDie sortierte Reihenfolge lautet:\n");
        if(a==0)
        {
            temp3=*(temp2);
            temp4=*(temp1);

            if(temp4>temp3)
                a=1;
            if(temp4<temp3)
                a=2;
            if(temp4==temp3)
```

```
                printf("Die Zeichenketten sind gleich\n");
            break;
        }

        if(a==1)
        {
            printf("%s\n",Zeichenkette2_gespeichert);
            printf("%s\n",Zeichenkette1_gespeichert);
            break;
        }

        if(a==2)
        {
            printf("%s\n",Zeichenkette1_gespeichert);
            printf("%s\n",Zeichenkette2_gespeichert);
            break;
        }


    } while(0);

    free(zeichenkette1_gespeichert);  //Freigeben des Speicherplatzes
    zeichenkette1_gespeichert=NULL;

    free(zeichenkette2_gespeichert);  //Freigeben des Speicherplatzes
    zeichenkette1_gespeichert=NULL;

    return 0;
}


//Einlesefunktion
char* einlesen (int j)
{
    int start,c;
    char zeichenkette[LAENGE],*pt=NULL;

    printf("Bitte geben sie eine Zeichenkette mit maximal %d Zeichen
        ein: ",LAENGE-1);
    // Einlesen einer beliebigen Zeichenkette mit Sonderzeichen
    for(start=0;(start<LAENGE-1) && ((c=getchar()) != EOF) &&c!='\n' ;start++)
    {
        zeichenkette[start]=(char)c;

    }
    //Hinzufügen eines Nullbytes an die letzte Stelle
    zeichenkette[start] = '\0';

    if(start==LAENGE-1 && !(c == EOF || c =='\n'))  //zu viele Zeichen
    {
        printf("Sie haben zu viele Zeichen eingeben. Diese koennen
            nicht beruecksichtigt werden\n");
        while(getchar()!='\n'); //übrige Zeichen Einlesen bis Enter
        printf("\tes konnte nur %s beruecksichtigt werden\n\n",zeichenkette);
    }


    //Speicheranforderung
    pt =malloc((start+1)*sizeof(char));

    if (pt!=NULL)
    {
        strcpy(pt,zeichenkette);
    }

    return pt;
}
```

23.7 Messdaten

23.7.1 Aufgabenstellung

Schreiben Sie ein Programm, das eine Messdatendatei, die Strom- und Spannungswerte enthält, ausliest und daraus folgende Kennwerte für jede Größe berechnet:

- Minimal- und Maximalwert,
- Gleichanteil (linearer Mittelwert),
- Effektivwert (geometrischer Mittelwert),
- Wirk- und Blindleistung.

Der Name der Datei soll als Kommandozeilenargument übergeben werden. Über die Angabe einer Option in der Kommandozeile sollen nur die Messdaten auf dem Bildschirm ausgegeben werden. Aufrufbeispiele für das Programm sind

- Berechnung und Ausgabe der Kennwerte: Aufgabe07.exe messdaten.txt
- Ausgabe der Messdatenpaare: Aufgabe07.exe messdaten.txt -print

Vor der Berechnung oder Ausgabe sollen alle Messwerte eingelesen werden. Auf die Daten soll über ein Array von Zeigern, die auf jeweils ein Messdatenpaar verweisen angesprochen werden. Nach dem letzten Datenpaar soll das nachfolgende Element ein Null-Pointer sein, um das Ende zu markieren. Die Datenstruktur könnte zum Beispiel wie folgt definiert werden:

```
typedef struct messwerte
{
 float spannung, strom;
}MESSWERTE;

MESSWERTE *daten[MAX ANZAHL];
```

Die Berechnung und Ausgabe der Kennwerte auf dem Bildschirm soll in einer eigens definierten Funktion realisiert werden. Die Ausgabe der Messwerte soll ebenfalls durch eine Funktion erfolgen. Dabei sollen die Werte tabellarisch auf dem Bildschirm seitenweise ausgeben werden (pro Ausgabeseite 25 Zeilen). Folgende Fehlersituationen sind zu berücksichtigen:

- Die Anzahl der Kommandozeilenargumente ist falsch.
- Die Messdatendatei lässt sich nicht öffnen.
- Beim Einlesen der Messdaten steht kein Speicherplatz mehr zur Verfügung.
- In der Messdatendatei stehen mehr Datenpaare als im Array gespeichert werden können.

Im Fehlerfall soll das Programm auf dem Bildschirm eine entsprechende Meldung ausgeben, ggf. bereitgestellten Speicher wieder freigeben und sich beenden.

23.7.2 Musterlösung

```
# include<stdio.h>
# include<string.h>
# include<stdlib.h>
# include<math.h>
# define MAX_ANZAHL 700
# define PI 3.14159265
```

```
//Prototyp der Struktur
typedef struct messwert
{
    float spannung, strom;
} MESSWERTE;

//Prototyp der Funktionen
int BerechnungAusgabe(MESSWERTE *daten[],int start);
int AusgabeMessdaten(MESSWERTE *daten[],int start);
void speicherfreigabe(MESSWERTE *daten[],int start);

//Hauptfunktion
//============================
================================================================================

int main(int argc, char *argv[])

{
    int  i=0;
    int start;
    MESSWERTE *daten [MAX_ANZAHL];    // Array von Zeigern
    float sp,str;//spannung,strom
    FILE *fp;

    if(argc==1)    // keine Parameter eingegeben, Fehler melden und
 Hilfestellung
    {
        printf("Ihre Eingabe  stimmt nicht!");
        printf("\n   Geben sie einen Namen fuer das zu oeffnende Argument an \n
  z.b.name.txt
                zum oeffnen und verarbeiten!");
        printf("\n   wenn der Inhalt gezeigt werden soll name.txt -print");
        return 1;
    }

    fp = fopen(argv[1],"r");    // Öffnen der Datei zum Lesen

    if(fp == NULL)        // wenn Datei nicht geöffnet werden konnte
 Fehlerhinweise und Ende
    {
        fprintf(stderr, "\nFehler beim Oeffnen der Datei %s\n",argv[1]);
        printf("Ihre Eingabe ist moeglicherweise falsch - beachten sie die
 Beispiele!");
        printf("\n   Geben sie einen Namen fuer das zu oeffnende Argument an \n
  z.B. name.txt zum oeffnen und
                            verarbeiten!");
        printf("\n   Ausgabe des Inhalts: name.txt -print");
        return 2;
    }

    if (argc==3)// filtern der Eingabe ob Argument belegt und wenn, ob richtig
        if(strcmp (argv[2],"-print")!=0)
        {
        printf("\n\nder Parameter %s ist falsch",argv[2]);
        printf("\n   Ausgabe des Inhalts: name.txt -print");
        return 3;
        }


    fprintf(stderr, "\nDatei %s wurde zum Lesen geoeffnet!\n\n",argv[1]);
    printf("%s;%s",argv[1],argv[2]);


    start=0;
    while(fscanf(fp,"%f;%f\n",&str,&sp)!=EOF && start < MAX_ANZAHL-1)    //
 Einlesen der Messreihe
```

```
    {

        // Speicherplatz anfordern
        if((daten[start] = malloc( sizeof (MESSWERTE))) == NULL)
        {
            fprintf(stderr,"Kein freier Speicher vorhanden.\n");
            fclose(fp);
            speicherfreigabe(daten,start);

            return -1;

        }
        // kopieren der Hilfsvariablen in das Array
        daten[start]->strom=str;
        daten[start]->spannung=sp;

        start++;
    }

    // anfügen des null pointers
    daten[start] = NULL;

    if (start >= MAX_ANZAHL)    // wenn mehr Messwerte als Speicheradressen
 vorhanden sind: Fehler
        printf("Beim Einlesen der Messdaten steht kein Speicherplatz im Array
 zur Verfuegung");
    fclose(fp);

    switch (argc) // Fallunterscheidung zwischen berechen und ausgaben der reihe
    {
    case 2:
        BerechnungAusgabe(daten,start);    // aufruf der rechen funktion


        break;

    case 3:
            //filtern der eingabe

            AusgabeMessdaten(daten,start);

        break;

    default: printf("\nIhre Eingabe wurde nicht akzeptiert, eventuell wurden zu
 viele Parameter eingegeben.");

        break;
    }

    speicherfreigabe(daten,start);

    return 0;
}


// Funktion zum Ermitteln der benötigten Daten aus der Messreihe und Berechnung
//=============================================================================

int BerechnungAusgabe(MESSWERTE *daten[],int start)
{
    double max_strom,max_spannung,min_strom,min_spannung;
    double u_gleichricht,i_gleichricht,u_effektiv, i_effektiv,p_wirk,p_blint;
    double max_spannung_sp1,
 max_spannung_sp2,max_strom_sp1,max_strom_sp2,cos_phi;
    float temp1,temp2,temp3,temp4;

    max_strom=0;
    max_spannung=0;
```

```
    min_strom=0;
    min_spannung=100000000;

    // Suchen von min- und max-Werten
    //==================
================================================================================

    for(i=0;i<start;i++)
    {
        if(max_strom<daten[i]->strom )
        {
            max_strom=daten[i]->strom;
        }

        if (max_spannung<daten[i]->spannung)
        {
            max_spannung=daten[i]->spannung;
        }
        if(min_strom>daten[i]->strom)
        {
            min_strom=daten[i]->strom;
        }
        if( min_spannung>daten[i]->spannung)
        {
            min_spannung=daten[i]->spannung;
        }

    }


    // Ermittlung von Daten zur Bestimmung des cos phi
    //=
================================================================================
    max_spannung_sp1=0;
    max_spannung_sp2=0;
    max_strom_sp2=0;
    max_strom_sp1=0;
    temp3=0;
    temp4=0;
    temp1=0;
    temp2=0;

    for(i=0;i<start-2;i++)  // Schleife zum Finden der Maxima von Strom und
 Spannung, sowie deren Abstaenden
    {
        if (d
aten[i]->spannung>daten[i+1]->spannung&&daten[i]->spannung>daten[i+2]->spannung)
        {
            if (daten[i]->spannung>daten[i-1]->spannung
 &&daten[i]->spannung>daten[i-2]->spannung)
                if(temp2==0 && temp1!=0)
                {
                    max_spannung_sp2=daten[i]->spannung;
                    temp2=(float)i;
                }
        }

        if (daten[i]->spannung>daten[i+1]->spannung&&daten[i+2]->spannung )
        {
            if (daten[i]->spannung>daten[i-1]->spannung && daten[i-2]->spannung)

                if(temp1==0)
                {
                    max_spannung_sp1=daten[i]->spannung;
                    temp1=(float)i;
                }
        }
```

```
        if(daten[i]->strom>daten[i+1]->strom&&daten[i]->strom>daten[i+2]->strom)
        {
            if
 (daten[i]->strom>daten[i-1]->strom&&daten[i]->strom>daten[i-2]->strom)
                if (temp4==0 && temp3!=0)
                {
                    max_strom_sp2=daten[i]->strom;
                    temp4=(float)i;
                }
        }

        if(daten[i]->strom>daten[i+1]->strom&&daten[i]->strom>daten[i+2]->strom)
        {
            if
 (daten[i]->strom>daten[i-1]->strom&&daten[i]->strom>daten[i-2]->strom)
                if (temp3==0)
                {
                    max_strom_sp1=daten[i]->strom;
                    temp3=(float)i;
                }
        }
    }

    // Berechnung der einzelnen Daten
    //===============
================================================================================
    // Berechnen des Gleichrichtwertes
    u_gleichricht=2/PI*max_spannung;
    i_gleichricht=2/PI*max_strom;


    // Berechnen des Effektivwertes
    u_effektiv=max_spannung/sqrt(2);

    i_effektiv=max_strom/sqrt(2);

    // Berechnung Phasenverschiebungswinkel
    cos_phi=(temp1-temp3)*360/(temp2-temp1);

    // Berechnung der Leistungsdaten
    p_wirk=u_effektiv*i_effektiv*cos(cos_phi*PI/180);
    p_blint=u_effektiv*i_effektiv*sin(cos_phi*PI/180);


    // Ausgabe der berechneten werte
    //===============
================================================================================


    printf("\n\nDie berechneten Werte fuer ihre Messdatenreihe lauten:");

    printf("\n\n=========================================");
    printf("\n \t\t||  Strom  ||  Spannung\t||");
    printf("\n=========================================");
    // Ausgabe Minima, Maxima
    printf("\n maxima\t\t|| %3.3lf A ||  %3.3lf V\t||",max_strom,max_spannung);
    printf("\n=========================================");
    printf("\n minima\t\t||%3.3lf A || %3.3lf V\t||",min_strom,min_spannung);
    printf("\n=========================================");
    // Ausgabe der Effektivwerte
    printf("\n Effektivwert\t|| %3.3lf A ||  %3.3lf
 V\t||",i_effektiv,u_effektiv);
    printf("\n=========================================");
    // Ausgabe der Gleichrichtwerte
    printf("\n Gleichrichtwert|| %3.3lf A ||  %3.3lf V
 \t||",i_gleichricht,u_gleichricht);
    printf("\n=========================================");
    // Ausgabe der Leistungsdaten
```

```
    printf("\n\n cos_phi:\t%3.3f Grad",cos_phi);
    printf(" \n Wirkleistung:\t  %3.3lf W\n Blindleistung:\t % 3.3lf
 VAR\n",p_wirk,p_blint);

    return 0;
}

//            Ausgabe der Messreihe
//=======================================================

int AusgabeMessdaten (MESSWERTE *daten[],int start)
{

    int i;


    printf("\n\nEs werden je Seite 25 Zeilen ausgegben\n Zum Weiterkommen
 <Enter> druecken. ;-)");
    printf("\n\tSpanung || Strom");

    for(i=0;i<start;i++)
    {
        if(i%25==0)
            getchar(); //Alle 25 Zeilen enter drücken

        printf("%4d  %8.4f  || %8.4f\n", i,daten[i]->spannung,daten[i]->strom);

    }
    return 0;
}

//            Funktion zum Freigeben des Speicherplatzes
//=========
================================================================================

void speicherfreigabe(MESSWERTE *daten[],int start)
{
    int start1;
    // Schleife zum Freigeben jeder einzelnden Arrayadresse
    for(start1=0;start1<start;start1++)
    {
        free(daten[start1]);
    }
}
```

24 Autoren

Edits	User
2	Ap0calypse[1]
11	Bastie[2]
4	Berni[3]
2	Biezl[4]
1	Boehm[5]
4	Borstel[6]
3	Buchfreund[7]
3	C.hahn[8]
1	Caotic[9]
3	Castagir[10]
2	Chatter[11]
1	Chirak[12]
1	Chloch[13]
2	ChrisiPK[14]
451	Daniel B[15]
1	Der Messer[16]
3	Derbeth[17]
53	Dirk Huenniger[18]
1	Eddi[19]
2	Enomil[20]
3	Floklk[21]

1 http://de.wikibooks.org/wiki/Benutzer:Ap0calypse
2 http://de.wikibooks.org/wiki/Benutzer:Bastie
3 http://de.wikibooks.org/wiki/Benutzer:Berni
4 http://de.wikibooks.org/wiki/Benutzer:Biezl
5 http://de.wikibooks.org/wiki/Benutzer:Boehm
6 http://de.wikibooks.org/wiki/Benutzer:Borstel
7 http://de.wikibooks.org/wiki/Benutzer:Buchfreund
8 http://de.wikibooks.org/wiki/Benutzer:C.hahn
9 http://de.wikibooks.org/wiki/Benutzer:Caotic
10 http://de.wikibooks.org/wiki/Benutzer:Castagir
11 http://de.wikibooks.org/wiki/Benutzer:Chatter
12 http://de.wikibooks.org/wiki/Benutzer:Chirak
13 http://de.wikibooks.org/wiki/Benutzer:Chloch
14 http://de.wikibooks.org/wiki/Benutzer:ChrisiPK
15 http://de.wikibooks.org/wiki/Benutzer:Daniel_B
16 http://de.wikibooks.org/wiki/Benutzer:Der_Messer
17 http://de.wikibooks.org/wiki/Benutzer:Derbeth
18 http://de.wikibooks.org/wiki/Benutzer:Dirk_Huenniger
19 http://de.wikibooks.org/wiki/Benutzer:Eddi
20 http://de.wikibooks.org/wiki/Benutzer:Enomil
21 http://de.wikibooks.org/wiki/Benutzer:Floklk

3	Florian Weber[22]
5	Geitost[23]
2	Gronau[24]
3	Hardy42[25]
11	Heuler06[26]
2	Hoo man[27]
1	InselFahrer[28]
31	Jack.van-dayk[29]
1	JackBot[30]
2	JackPotte[31]
1	Jackson[32]
2	Jan[33]
8	Jdaly[34]
21	Juetho[35]
2	Kai Burghardt[36]
1	Klartext[37]
131	Klaus Eifert[38]
34	Matthias M.[39]
3	Merkel[40]
5	MichaelFrey[41]
7	MichaelFreyTool[42]
4	Mik[43]
1	Mjchael[44]
3	Moolsan[45]
2	NeuerNutzer2009[46]

22 http://de.wikibooks.org/wiki/Benutzer:Florian_Weber
23 http://de.wikibooks.org/wiki/Benutzer:Geitost
24 http://de.wikibooks.org/wiki/Benutzer:Gronau
25 http://de.wikibooks.org/wiki/Benutzer:Hardy42
26 http://de.wikibooks.org/wiki/Benutzer:Heuler06
27 http://de.wikibooks.org/wiki/Benutzer:Hoo_man
28 http://de.wikibooks.org/wiki/Benutzer:InselFahrer
29 http://de.wikibooks.org/wiki/Benutzer:Jack.van-dayk
30 http://de.wikibooks.org/wiki/Benutzer:JackBot
31 http://de.wikibooks.org/wiki/Benutzer:JackPotte
32 http://de.wikibooks.org/wiki/Benutzer:Jackson
33 http://de.wikibooks.org/wiki/Benutzer:Jan
34 http://de.wikibooks.org/wiki/Benutzer:Jdaly
35 http://de.wikibooks.org/wiki/Benutzer:Juetho
36 http://de.wikibooks.org/wiki/Benutzer:Kai_Burghardt
37 http://de.wikibooks.org/wiki/Benutzer:Klartext
38 http://de.wikibooks.org/wiki/Benutzer:Klaus_Eifert
39 http://de.wikibooks.org/wiki/Benutzer:Matthias_M.
40 http://de.wikibooks.org/wiki/Benutzer:Merkel
41 http://de.wikibooks.org/wiki/Benutzer:MichaelFrey
42 http://de.wikibooks.org/wiki/Benutzer:MichaelFreyTool
43 http://de.wikibooks.org/wiki/Benutzer:Mik
44 http://de.wikibooks.org/wiki/Benutzer:Mjchael
45 http://de.wikibooks.org/wiki/Benutzer:Moolsan
46 http://de.wikibooks.org/wiki/Benutzer:NeuerNutzer2009

1 Nowotoj[47]
2 Obstriegel[48]
1 PaMaRo[49]
2 Polluks[50]
11 Prog[51]
1 Progman[52]
3 Quiethoo[53]
2 Raymontag[54]
31 Revolus[55]
2 Sae1962[56]
142 Stefan Kögl[57]
5 Stefan-Xp[58]
1 Stephan Kulla[59]
12 ThePacker[60]
1 Thirafydion[61]
10 Thomas-Chris[62]
2 Thopre[63]
2 Tigertech8[64]
4 Tobi84[65]
1 Trixium[66]
2 UbuntuPeter[67]
1 WikiBookPhil[68]
28 WissensDürster[69]
2 Wizzar[70]
1 Wolfgang1018[71]

47 http://de.wikibooks.org/wiki/Benutzer:Nowotoj
48 http://de.wikibooks.org/wiki/Benutzer:Obstriegel
49 http://de.wikibooks.org/wiki/Benutzer:PaMaRo
50 http://de.wikibooks.org/wiki/Benutzer:Polluks
51 http://de.wikibooks.org/wiki/Benutzer:Prog
52 http://de.wikibooks.org/wiki/Benutzer:Progman
53 http://de.wikibooks.org/wiki/Benutzer:Quiethoo
54 http://de.wikibooks.org/wiki/Benutzer:Raymontag
55 http://de.wikibooks.org/wiki/Benutzer:Revolus
56 http://de.wikibooks.org/wiki/Benutzer:Sae1962
57 http://de.wikibooks.org/wiki/Benutzer:Stefan_K%25C3%25B6gl
58 http://de.wikibooks.org/wiki/Benutzer:Stefan-Xp
59 http://de.wikibooks.org/wiki/Benutzer:Stephan_Kulla
60 http://de.wikibooks.org/wiki/Benutzer:ThePacker
61 http://de.wikibooks.org/wiki/Benutzer:Thirafydion
62 http://de.wikibooks.org/wiki/Benutzer:Thomas-Chris
63 http://de.wikibooks.org/wiki/Benutzer:Thopre
64 http://de.wikibooks.org/wiki/Benutzer:Tigertech8
65 http://de.wikibooks.org/wiki/Benutzer:Tobi84
66 http://de.wikibooks.org/wiki/Benutzer:Trixium
67 http://de.wikibooks.org/wiki/Benutzer:UbuntuPeter
68 http://de.wikibooks.org/wiki/Benutzer:WikiBookPhil
69 http://de.wikibooks.org/wiki/Benutzer:WissensD%25C3%25BCrster
70 http://de.wikibooks.org/wiki/Benutzer:Wizzar
71 http://de.wikibooks.org/wiki/Benutzer:Wolfgang1018

9 Worker[72]
25 Younix[73]
6 Yuuki Mayuki[74]
3 ☒☒☒☒ robot[75]

72 http://de.wikibooks.org/wiki/Benutzer:Worker
73 http://de.wikibooks.org/wiki/Benutzer:Younix
74 http://de.wikibooks.org/wiki/Benutzer:Yuuki_Mayuki
75 http://de.wikibooks.org/wiki/Benutzer:%25E3%2582%25BF%25E3%2583%2581%25E3%2582%25B3%25E3%2583%259E_robot

Abbildungsverzeichnis

- GFDL: Gnu Free Documentation License. http://www.gnu.org/licenses/fdl.html
- cc-by-sa-3.0: Creative Commons Attribution ShareAlike 3.0 License. http://creativecommons.org/licenses/by-sa/3.0/
- cc-by-sa-2.5: Creative Commons Attribution ShareAlike 2.5 License. http://creativecommons.org/licenses/by-sa/2.5/
- cc-by-sa-2.0: Creative Commons Attribution ShareAlike 2.0 License. http://creativecommons.org/licenses/by-sa/2.0/
- cc-by-sa-1.0: Creative Commons Attribution ShareAlike 1.0 License. http://creativecommons.org/licenses/by-sa/1.0/
- cc-by-2.0: Creative Commons Attribution 2.0 License. http://creativecommons.org/licenses/by/2.0/
- cc-by-2.0: Creative Commons Attribution 2.0 License. http://creativecommons.org/licenses/by/2.0/deed.en
- cc-by-2.5: Creative Commons Attribution 2.5 License. http://creativecommons.org/licenses/by/2.5/deed.en
- cc-by-3.0: Creative Commons Attribution 3.0 License. http://creativecommons.org/licenses/by/3.0/deed.en
- GPL: GNU General Public License. http://www.gnu.org/licenses/gpl-2.0.txt
- LGPL: GNU Lesser General Public License. http://www.gnu.org/licenses/lgpl.html
- PD: This image is in the public domain.
- ATTR: The copyright holder of this file allows anyone to use it for any purpose, provided that the copyright holder is properly attributed. Redistribution, derivative work, commercial use, and all other use is permitted.
- EURO: This is the common (reverse) face of a euro coin. The copyright on the design of the common face of the euro coins belongs to the European Commission. Authorised is reproduction in a format without relief (drawings, paintings, films) provided they are not detrimental to the image of the euro.
- LFK: Lizenz Freie Kunst. http://artlibre.org/licence/lal/de
- CFR: Copyright free use.

- EPL: Eclipse Public License. `http://www.eclipse.org/org/documents/epl-v10.php`

Copies of the GPL, the LGPL as well as a GFDL are included in chapter Licenses[76]. Please note that images in the public domain do not require attribution. You may click on the image numbers in the following table to open the webpage of the images in your webbrower.

76 Kapitel 25 auf Seite 247

1	Bdk, Daniel B, Dirk Huenniger, Heuler06	
2	Daniel B, Dirk Huenniger, Heuler06	
3	Daniel B[77]	CC-BY-SA-3.0
4	Daniel B[78]	CC-BY-SA-3.0
5	Daniel B[79]	CC-BY-SA-3.0
6	Daniel B[80]	CC-BY-SA-3.0
7	Stefan-Xp[81]	CC-BY-SA-3.0

77 http://en.wikibooks.org/wiki/de:Benutzer:Daniel_B
78 http://en.wikibooks.org/wiki/de:Benutzer:Daniel_B
79 http://en.wikibooks.org/wiki/de:Benutzer:Daniel_B
80 http://en.wikibooks.org/wiki/de:Benutzer:Daniel_B
81 http://commons.wikimedia.org/wiki/User:Stefan-Xp

25 Licenses

25.1 GNU GENERAL PUBLIC LICENSE

Version 3, 29 June 2007

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed. Preamble

The GNU General Public License is a free, copyleft license for software and other kinds of works.

The licenses for most software and other practical works are designed to take away your freedom to share and change the works. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change all versions of a program–to make sure it remains free software for all its users. We, the Free Software Foundation, use the GNU General Public License for most of our software; it applies also to any other work released this way by its authors. You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for them if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs, and that you know you can do these things.

To protect your rights, we need to prevent others from denying you these rights or asking you to surrender the rights. Therefore, you have certain responsibilities if you distribute copies of the software, or if you modify it: responsibilities to respect the freedom of others.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must pass on to the recipients the same freedoms that you received. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

Developers that use the GNU GPL protect your rights with two steps: (1) assert copyright on the software, and (2) offer you this License giving you legal permission to copy, distribute and/or modify it.

For the developers' and authors' protection, the GPL clearly explains that there is no warranty for this free software. For both users' and authors' sake, the GPL requires that modified versions be marked as changed, so that their problems will not be attributed erroneously to authors of previous versions.

Some devices are designed to deny users access to install or run modified versions of the software inside them, although the manufacturer can do so. This is fundamentally incompatible with the aim of protecting users' freedom to change the software. The systematic pattern of such abuse occurs in the area of products for individuals to use, which is precisely where it is most unacceptable. Therefore, we have designed this version of the GPL to prohibit the practice for those products. If such problems arise substantially in other domains, we stand ready to extend this provision to those domains in future versions of the GPL, as needed to protect the freedom of users.

Finally, every program is threatened constantly by software patents. States should not allow patents to restrict development and use of software on general-purpose computers, but in those that do, we wish to avoid the special danger that patents applied to a free program could make it effectively proprietary. To prevent this, the GPL assures that patents cannot be used to render the program non-free.

The precise terms and conditions for copying, distribution and modification follow. TERMS AND CONDITIONS 0. Definitions.

"This License" refers to version 3 of the GNU General Public License.

"Copyright" also means copyright-like laws that apply to other kinds of works, such as semiconductor masks.

"The Program" refers to any copyrightable work licensed under this License. Each licensee is addressed as "you". "Licensees" and "recipients" may be individuals or organizations.

To "modify" a work means to copy from or adapt all or part of the work in a fashion requiring copyright permission, other than the making of an exact copy. The resulting work is called a "modified version" of the earlier work or a work "based on" the earlier work.

A "covered work" means either the unmodified Program or a work based on the Program.

To "propagate" a work means to do anything with it that, without permission, would make you directly or secondarily liable for infringement under applicable copyright law, except executing it on a computer or modifying a private copy. Propagation includes copying, distribution (with or without modification), making available to the public, and in some countries other activities as well.

To "convey" a work means any kind of propagation that enables other parties to make or receive copies. Mere interaction with a user through a computer network, with no transfer of a copy, is not conveying.

An interactive user interface displays "Appropriate Legal Notices" to the extent that it includes a convenient and prominently visible feature that (1) displays an appropriate copyright notice, and (2) tells the user that there is no warranty for the work (except to the extent that warranties are provided), that licensees may convey the work under this License, and how to view a copy of this License. If the interface presents a list of user commands or options, such as a menu, a prominent item in the list meets this criterion. 1. Source Code.

The "source code" for a work means the preferred form of the work for making modifications to it. "Object code" means any non-source form of a work.

A "Standard Interface" means an interface that either is an official standard defined by a recognized standards body, or, in the case of interfaces specified for a particular programming language, one that is widely used among developers working in that language.

The "System Libraries" of an executable work include anything, other than the work as a whole, that (a) is included in the normal form of packaging a Major Component, but which is not part of that Major Component, and (b) serves only to enable use of the work with that Major Component, or to implement a Standard Interface for which an implementation is available to the public in source code form. A "Major Component", in this context, means a major essential component (kernel, window system, and so on) of the specific operating system (if any) on which the executable work runs, or a compiler used to produce the work, or an object code interpreter used to run it.

The "Corresponding Source" for a work in object code form means all the source code needed to generate, install, and (for an executable work) run the object code and to modify the work, including scripts to control those activities. However, it does not include the work's System Libraries, or general-purpose tools or generally available free programs which are used unmodified in performing those activities but which are not part of the work. For example, Corresponding Source includes interface definition files associated with source files for the work, and the source code for shared libraries and dynamically linked subprograms that the work is specifically designed to require, such as by intimate data communication or control flow between those subprograms and other parts of the work.

The Corresponding Source need not include anything that users can regenerate automatically from other parts of the Corresponding Source.

The Corresponding Source for a work in source code form is that same work. 2. Basic Permissions.

All rights granted under this License are granted for the term of copyright on the Program, and are irrevocable provided the stated conditions are met. This License explicitly affirms your unlimited permission to run the unmodified Program. The output from running a covered work is covered by this License only if the output, given its content, constitutes a covered work. This License acknowledges your rights of fair use or other equivalent, as provided by copyright law.

You may make, run and propagate covered works that you do not convey, without conditions so long as your license otherwise remains in force. You may convey covered works to others for the sole purpose of having them make modifications exclusively for you, or provide you with facilities for running those works, provided that you comply with the terms of this License in conveying all material for which you do not control copyright. Those thus making or running the covered works for you must do so exclusively on your behalf, under your direction and control, on terms that prohibit them from making any copies of your copyrighted material outside their relationship with you.

Conveying under any other circumstances is permitted solely under the conditions stated below. Sublicensing is not allowed; section 10 makes it unnecessary. 3. Protecting Users' Legal Rights From Anti-Circumvention Law.

No covered work shall be deemed part of an effective technological measure under any applicable law fulfilling obligations under article 11 of the WIPO copyright treaty adopted on 20 December 1996, or similar laws prohibiting or restricting circumvention of such measures.

When you convey a covered work, you waive any legal power to forbid circumvention of technological measures to the extent such circumvention is effected by exercising rights under this License with respect to the covered work, and you disclaim any intention to limit operation or modification of the work as a means of enforcing, against the work's users, your or third parties' legal rights to forbid circumvention of technological measures. 4. Conveying Verbatim Copies.

You may convey verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice; keep intact all notices stating that this License and any non-permissive terms added in accord with section 7 apply to the code; keep intact all notices of the absence of any warranty; and give all recipients a copy of this License along with the Program.

You may charge any price or no price for each copy that you convey, and you may offer support or warranty protection for a fee. 5. Conveying Modified Source Versions.

You may convey a work based on the Program, or the modifications to produce it from the Program, in the form of source code under the terms of section 4, provided that you also meet all of these conditions:

* a) The work must carry prominent notices stating that you modified it, and giving a relevant date. * b) The work must carry prominent notices stating that it is released under this License and any conditions added under section 7. This requirement modifies the requirement in section 4 to "keep intact all notices". * c) You must license the entire work, as a whole, under this License to anyone who comes into possession of a copy. This License will therefore apply, along with any applicable section 7 additional terms, to the whole of the work, and all its parts, regardless of how they are packaged. This License gives no permission to license the work in any other way, but it does not invalidate such permission if you have separately received it. * d) If the work has interactive user interfaces, each must display Appropriate Legal Notices; however, if the Program has interactive interfaces that do not display Appropriate Legal Notices, your work need not make them do so.

A compilation of a covered work with other separate and independent works, which are not by their nature extensions of the covered work, and which are not combined with it such as to form a larger program, in or on a volume of a storage or distribution medium, is called an "aggregate" if the compilation and its resulting copyright are not used to limit the access or legal rights of the compilation's users beyond what the individual works permit. Inclusion of a covered work in an aggregate does not cause this License to apply to the other parts of the aggregate. 6. Conveying Non-Source Forms.

You may convey a covered work in object code form under the terms of sections 4 and 5, provided that you also convey the machine-readable Corresponding Source under the terms of this License, in one of these ways:

* a) Convey the object code in, or embodied in, a physical product (including a physical distribution medium), accompanied by the Corresponding Source fixed on a durable physical medium customarily used for software interchange. * b) Convey the object code in, or embodied in, a physical product (including a physical distribution medium), accompanied by a written offer, valid for at least three years and valid for as long as you offer spare parts or customer support for that product model, to give anyone who possesses the object code either (1) a copy of the Corresponding Source for all the software in the product that is covered by this License, on a durable physical medium customarily used for software interchange, for a price no more than your reasonable cost of physically performing this conveying of source, or (2) access to copy the Corresponding Source from a network server at no charge. * c) Convey individual copies of the object code with a copy of the written offer to provide the Corresponding Source. This alternative is allowed only occasionally and noncommercially, and only if you received the object code with such an offer, in accord with subsection 6b. * d) Convey the object code by offering access from a designated place (gratis or for a charge), and offer equivalent access to the Corresponding Source in the same way through the same place at no further charge. You need not require recipients to copy the Corresponding Source along with the object code. If the place to copy the object code is a network server, the Corresponding Source may be on a different server (operated by you or a third party) that supports equivalent copying facilities, provided you maintain clear directions next to the object code saying where to find the Corresponding Source. Regardless of what server hosts the Corresponding Source, you remain obligated to ensure that it is available for as long as needed to satisfy these requirements. * e) Convey the object code using peer-to-peer transmission, provided you inform other peers where the object code and Corresponding Source of the work are being offered to the general public at no charge under subsection 6d.

A separable portion of the object code, whose source code is excluded from the Corresponding Source as a System Library, need not be included in conveying the object code work.

A "User Product" is either (1) a "consumer product", which means any tangible personal property which is normally used for personal, family, or household purposes, or (2) anything designed or sold for incorporation into a dwelling. In determining whether a product is a consumer product, doubtful cases shall be resolved in favor of coverage. For a particular product received by a particular user, "normally used" refers to a typical or common use of that class of product, regardless of the status of the particular user or of the way in which the particular user actually uses, or expects or is expected to use, the product. A product is a consumer product regardless of whether the product has substantial commercial, industrial or non-consumer uses, unless such uses represent the only significant mode of use of the product.

"Installation Information" for a User Product means any methods, procedures, authorization keys, or other information required to install and execute modified versions of a covered work in that User Product from a modified version of its Corresponding Source. The information must suffice to ensure that the continued functioning of the modified object code is in no case prevented or interfered with solely because modification has been made.

If you convey an object code work under this section in, or with, or specifically for use in, a User Product, and the conveying occurs as part of a transaction in which the right of possession and use of the User Product is transferred to the recipient in perpetuity or for a fixed term (regardless of how the transaction is characterized), the Corresponding Source conveyed under this section must be accompanied by the Installation Information. But this requirement does not apply if neither you nor any third party retains the ability to install modified object code on the User Product (for example, the work has been installed in ROM).

The requirement to provide Installation Information does not include a requirement to continue to provide support service, warranty, or updates for a work that has been modified or installed by the recipient, or for the User Product in which it has been modified or installed. Access to a network may be denied when the modification itself materially and adversely affects the operation of the network or violates the rules and protocols for communication across the network.

Corresponding Source conveyed, and Installation Information provided, in accord with this section must be in a format that is publicly documented (and with an implementation available to the public in source code form), and must require no special password or key for unpacking, reading or copying. 7. Additional Terms.

"Additional permissions" are terms that supplement the terms of this License by making exceptions from one or more of its conditions. Additional permissions that are applicable to the entire Program shall be treated as though they were included in this License, to the extent that they are valid under applicable law. If additional permissions apply only to part of the Program, that part may be used separately under those permissions, but the entire Program remains governed by this License without regard to the additional permissions.

When you convey a copy of a covered work, you may at your option remove any additional permissions from that copy, or from any part of it. (Additional permissions may be written to require their own removal in certain cases when you modify the work.) You may place additional permissions on material, added by you to a covered work, for which you have or can give appropriate copyright permission.

Notwithstanding any other provision of this License, for material you add to a covered work, you may (if authorized by the copyright holders of that material) supplement the terms of this License with terms:

* a) Disclaiming warranty or limiting liability differently from the terms of sections 15 and 16 of this License; or * b) Requiring preservation of specified reasonable legal notices or author attributions in that material or in the Appropriate Legal Notices displayed by works containing it; or * c) Prohibiting misrepresentation of the origin of that material, or requiring that modified versions of such material be marked in reasonable ways as different from the original version; or * d) Limiting the use for publicity purposes of names of licensors or authors of the material; or * e) Declining to grant rights under trademark law for use of some trade names, trademarks, or service marks; or * f) Requiring indemnification of licensors and authors of that material by anyone who conveys the material (or modified versions of it) with contractual assumptions of liability to the recipient, for any liability that these contractual assumptions directly impose on those licensors and authors.

All other non-permissive additional terms are considered "further restrictions" within the meaning of section 10. If the Program as you received it, or any part of it, contains a notice stating that it is governed by this License along with a term that is a further restriction, you may remove that term. If a license document contains a further restriction but permits relicensing or conveying under this License, you may add to a covered work material governed by the terms of that license document, provided that the further restriction does not survive such relicensing or conveying.

If you add terms to a covered work in accord with this section, you must place, in the relevant source files, a statement of the additional terms that apply to those files, or a notice indicating where to find the applicable terms.

Additional terms, permissive or non-permissive, may be stated in the form of a separately written license, or stated as exceptions; the above requirements apply either way. 8. Termination.

You may not propagate or modify a covered work except as expressly provided under this License. Any attempt otherwise to propagate or modify it is void, and will automatically terminate your rights under this License (including any patent licenses granted under the third paragraph of section 11).

However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, you do not qualify to receive new licenses for the same material under section 10. 9. Acceptance Not Required for Having Copies.

You are not required to accept this License in order to receive or run a copy of the Program. Ancillary propagation of a covered work occurring solely as a consequence of using peer-to-peer transmission to receive a copy likewise does not require acceptance. However, nothing other than this License grants you permission to propagate or modify any covered work. These actions infringe copyright if you do not accept this License. Therefore, by modifying or propagating a covered work, you indicate your acceptance of this License to do so. 10. Automatic Licensing of Downstream Recipients.

Each time you convey a covered work, the recipient automatically receives a license from the original licensors, to run, modify and propagate that work, subject to this License. You are not responsible for enforcing compliance by third parties with this License.

An "entity transaction" is a transaction transferring control of an organization, or substantially all assets of one, or subdividing an organization, or merging organizations. If propagation of a covered work results from an entity transaction, each party to that transaction who receives a copy of the work also receives whatever licenses to the work the party's predecessor in interest had or could give under the previous paragraph, plus a right to possession of the Corresponding Source of the work from the predecessor in interest, if the predecessor has it or can get it with reasonable efforts.

You may not impose any further restrictions on the exercise of the rights granted or affirmed under this License. For example, you may not impose a license fee, royalty, or other charge for exercise of rights granted under this License, and you may not initiate litigation (including a cross-claim or counterclaim in a lawsuit) alleging that any patent claim is infringed by making, using, selling, offering for sale, or importing the Program or any portion of it. 11. Patents.

A "contributor" is a copyright holder who authorizes use under this License of the Program or a work on which the Program is based. The work thus licensed is called the contributor's "contributor version".

A contributor's "essential patent claims" are all patent claims owned or controlled by the contributor, whether already acquired or hereafter acquired, that would be infringed by some manner, permitted by this License, of making, using, or selling its contributor version, but do not include claims that would be infringed only as a consequence of further modification of the contributor version. For purposes of this definition, "control" includes the right to grant patent sublicenses in a manner consistent with the requirements of this License.

Each contributor grants you a non-exclusive, worldwide, royalty-free patent license under the contributor's essential patent claims, to make, use, sell, offer for sale, import and otherwise run, modify and propagate the contents of its contributor version.

In the following three paragraphs, a "patent license" is any express agreement or commitment, however denominated, not to enforce a patent (such as an express permission to practice a patent or covenant not to sue for patent infringement). To "grant" such a patent license to a party means to make such an agreement or commitment not to enforce a patent against the party.

If you convey a covered work, knowingly relying on a patent license, and the Corresponding Source of the work is not available for anyone to copy, free of charge and under the terms of this License, through a publicly available network server or other readily accessible means, then you must either (1) cause the Corresponding Source to be so available, or (2) arrange to deprive yourself of the benefit of the patent license for this particular work, or (3) arrange, in a manner consistent with the requirements of this License, to extend the patent license to downstream recipients. "Knowingly relying" means you have actual knowledge that, but for the patent license, your conveying the covered work in a country, or your recipient's use of the covered work in a country, would infringe one or more identifiable patents in that country that you have reason to believe are valid.

If, pursuant to or in connection with a single transaction or arrangement, you convey, or propagate by procuring conveyance of, a covered work, and grant a patent license to some of the parties receiving the covered work authorizing them to use, propagate, modify or convey a specific copy of the covered work, then the patent license you grant is automatically extended to all recipients of the covered work and works based on it.

A patent license is "discriminatory" if it does not include within the scope of its coverage, prohibits the exercise of, or is conditioned on the non-exercise of one or more of the rights that are specifically granted under this License. You may not convey a covered work if you are a party to an arrangement with a third party that is in the business of distributing software, under which you make payment to the third party based on the extent of your activity of conveying the work, and under which the third party grants, to any of the parties who would receive the covered work from you, a discriminatory patent license (a) in connection with copies of the covered work conveyed by you (or copies made from those copies), or (b) primarily for and in connection with specific products or compilations that contain the covered work, unless you entered into that arrangement, or that patent license was granted, prior to 28 March 2007.

Nothing in this License shall be construed as excluding or limiting any implied license or other defenses to infringement that may otherwise be available to you under applicable patent law. 12. No Surrender of Others' Freedom.

If conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot convey a covered work so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not convey it at all. For example, if you agree to terms that obligate you to collect a royalty for further conveying from those to whom you convey the Program, the only way you could satisfy both those terms and this License would be to refrain entirely from

conveying the Program. 13. Use with the GNU Affero General Public License.

Notwithstanding any other provision of this License, you have permission to link or combine any covered work with a work licensed under version 3 of the GNU Affero General Public License into a single combined work, and to convey the resulting work. The terms of this License will continue to apply to the part which is the covered work, but the special requirements of the GNU Affero General Public License, section 13, concerning interaction through a network will apply to the combination as such. 14. Revised Versions of this License.

The Free Software Foundation may publish revised and/or new versions of the GNU General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Program specifies that a certain numbered version of the GNU General Public License "or any later version" applies to it, you have the option of following the terms and conditions either of that numbered version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of the GNU General Public License, you may choose any version ever published by the Free Software Foundation.

If the Program specifies that a proxy can decide which future versions of the GNU General Public License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Program.

Later license versions may give you additional or different permissions. However, no additional obligations are imposed on any author or copyright holder as a result of your choosing to follow a later version. 15. Disclaimer of Warranty.

THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION. 16. Limitation of Liability.

IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MODIFIES AND/OR CONVEYS THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. 17. Interpretation of Sections 15 and 16.

If the disclaimer of warranty and limitation of liability provided above cannot be given local legal effect according to their terms, reviewing courts shall apply local law that most closely approximates an absolute waiver of all civil liability in connection with the Program, unless a warranty or assumption of liability accompanies a copy of the Program in return for a fee.

END OF TERMS AND CONDITIONS How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively state the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

<one line to give the program's name and a brief idea of what it does.> Copyright (C) <year> <name of author>

This program is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program. If not, see <http://www.gnu.org/licenses/>.

Also add information on how to contact you by electronic and paper mail.

If the program does terminal interaction, make it output a short notice like this when it starts in an interactive mode:

<program> Copyright (C) <year> <name of author> This program comes with ABSOLUTELY NO WARRANTY; for details type 'show w'. This is free software, and you are welcome to redistribute it under certain conditions; type 'show c' for details.

The hypothetical commands 'show w' and 'show c' should show the appropriate parts of the General Public License. Of course, your program's commands might be different; for a GUI interface, you would use an "about box".

You should also get your employer (if you work as a programmer) or school, if any, to sign a "copyright disclaimer" for the program, if necessary. For more information on this, and how to apply and follow the GNU GPL, see <http://www.gnu.org/licenses/>.

The GNU General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Lesser General Public License instead of this License. But first, please read <http://www.gnu.org/philosophy/why-not-lgpl.html>.

25.2 GNU Free Documentation License

Version 1.3, 3 November 2008

Copyright © 2000, 2001, 2002, 2007, 2008 Free Software Foundation, Inc. <http://fsf.org/>

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed. 0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference. 1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

The "publisher" means any person or entity that distributes copies of the Document to the public.

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgements", "Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License. 2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies. 3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document. 4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

* A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission. * B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement. * C. State on the Title page the name of the publisher of the Modified Version, as the publisher. * D. Preserve all the copyright notices of the Document. * E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices. * F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below. * G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice. * H. Include an unaltered copy of this License. * I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence. * J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission. * K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein. * L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles. * M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version. * N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section. * O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties—for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version. 5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements". 6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document. 7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate. 8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title. 9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, or distribute it is void, and will automatically terminate your rights under this License.

However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, receipt of a copy of some or all of the same material does not give you any rights to use it. 10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation. If the Document specifies that a proxy can decide which future versions of this License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Document. 11. RELICENSING

"Massive Multiauthor Collaboration Site" (or "MMC Site") means any World Wide Web server that publishes copyrightable works and also provides prominent facilities for anybody to edit those works. A public wiki that anybody can edit is an example of such a server. A "Massive Multiauthor Collaboration" (or "MMC") contained in the site means any set of copyrightable works thus published on the MMC site.

"CC-BY-SA" means the Creative Commons Attribution-Share Alike 3.0 license published by Creative Commons Corporation, a not-for-profit corporation with a principal place of business in San Francisco, California, as well as future copyleft versions of that license published by that same organization.

"Incorporate" means to publish or republish a Document, in whole or in part, as part of another Document.

An MMC is "eligible for relicensing" if it is licensed under this License, and if all works that were first published under this License somewhere other than this MMC, and subsequently incorporated in whole or in part into the MMC, (1) had no cover texts or invariant sections, and (2) were thus incorporated prior to November 1, 2008.

The operator of an MMC Site may republish an MMC contained in the site under CC-BY-SA on the same site at any time before August 1, 2009, provided the MMC is eligible for relicensing. ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

Copyright (C) YEAR YOUR NAME. Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.3 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled "GNU Free Documentation License".

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with ... Texts." line with this:

with the Invariant Sections being LIST THEIR TITLES, with the Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST.

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

25.3 GNU Lesser General Public License

GNU LESSER GENERAL PUBLIC LICENSE

Version 3, 29 June 2007

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

This version of the GNU Lesser General Public License incorporates the terms and conditions of version 3 of the GNU General Public License, supplemented by the additional permissions listed below. 0. Additional Definitions.

As used herein, “this License” refers to version 3 of the GNU Lesser General Public License, and the “GNU GPL” refers to version 3 of the GNU General Public License.

“The Library” refers to a covered work governed by this License, other than an Application or a Combined Work as defined below.

An “Application” is any work that makes use of an interface provided by the Library, but which is not otherwise based on the Library. Defining a subclass of a class defined by the Library is deemed a mode of using an interface provided by the Library.

A “Combined Work” is a work produced by combining or linking an Application with the Library. The particular version of the Library with which the Combined Work was made is also called the “Linked Version”.

The “Minimal Corresponding Source” for a Combined Work means the Corresponding Source for the Combined Work, excluding any source code for portions of the Combined Work that, considered in isolation, are based on the Application, and not on the Linked Version.

The “Corresponding Application Code” for a Combined Work means the object code and/or source code for the Application, including any data and utility programs needed for reproducing the Combined Work from the Application, but excluding the System Libraries of the Combined Work. 1. Exception to Section 3 of the GNU GPL.

You may convey a covered work under sections 3 and 4 of this License without being bound by section 3 of the GNU GPL. 2. Conveying Modified Versions.

If you modify a copy of the Library, and, in your modifications, a facility refers to a function or data to be supplied by an Application that uses the facility (other than as an argument passed when the facility is invoked), then you may convey a copy of the modified version:

* a) under this License, provided that you make a good faith effort to ensure that, in the event an Application does not supply the function or data, the facility still operates, and performs whatever part of its purpose remains meaningful, or * b) under the GNU GPL, with none of the additional permissions of this License applicable to that copy.

3. Object Code Incorporating Material from Library Header Files.

The object code form of an Application may incorporate material from a header file that is part of the Library. You may convey such object code under terms of your choice, provided that, if the incorporated material is not limited to numerical parameters, data structure layouts and accessors, or small macros, inline functions and templates (ten or fewer lines in length), you do both of the following:

* a) Give prominent notice with each copy of the object code that the Library is used in it and that the Library and its use are covered by this License. * b) Accompany the object code with a copy of the GNU GPL and this license document.

4. Combined Works.

You may convey a Combined Work under terms of your choice that, taken together, effectively do not restrict modification of the portions of the Library contained in the Combined Work and reverse engineering for debugging such modifications, if you also do each of the following:

* a) Give prominent notice with each copy of the Combined Work that the Library is used in it and that the Library and its use are covered by this License. * b) Accompany the Combined Work with a copy of the GNU GPL and this license document. * c) For a Combined Work that displays copyright notices during execution, include the copyright notice for the Library among these notices, as well as a reference directing the user to the copies of the GNU GPL and this license document. * d) Do one of the following: o 0) Convey the Minimal Corresponding Source under the terms of this License, and the Corresponding Application Code in a form suitable for, and under terms that permit, the user to recombine or relink the Application with a modified version of the Linked Version to produce a modified Combined Work, in the manner specified by section 6 of the GNU GPL for conveying Corresponding Source. o 1) Use a suitable shared library mechanism for linking with the Library. A suitable mechanism is one that (a) uses at run time a copy of the Library already present on the user’s computer system, and (b) will operate properly with a modified version of the Library that is interface-compatible with the Linked Version. * e) Provide Installation Information, but only if you would otherwise be required to provide such information under section 6 of the GNU GPL, and only to the extent that such information is necessary to install and execute a modified version of the Combined Work produced by recombining or relinking the Application with a modified version of the Linked Version. (If you use option 4d0, the Installation Information must accompany the Minimal Corresponding Source and Corresponding Application Code. If you use option 4d1, you must provide the Installation Information in the manner specified by section 6 of the GNU GPL for conveying Corresponding Source.)

5. Combined Libraries.

You may place library facilities that are a work based on the Library side by side in a single library together with other library facilities that are not Applications and are not covered by this License, and convey such a combined library under terms of your choice, if you do both of the following:

* a) Accompany the combined library with a copy of the same work based on the Library, uncombined with any other library facilities, conveyed under the terms of this License. * b) Give prominent notice with the combined library that part of it is a work based on the Library, and explaining where to find the accompanying uncombined form of the same work.

6. Revised Versions of the GNU Lesser General Public License.

The Free Software Foundation may publish revised and/or new versions of the GNU Lesser General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Library as you received it specifies that a certain numbered version of the GNU Lesser General Public License “or any later version” applies to it, you have the option of following the terms and conditions either of that published version or of any later version published by the Free Software Foundation. If the Library as you received it does not specify a version number of the GNU Lesser General Public License, you may choose any version of the GNU Lesser General Public License ever published by the Free Software Foundation.

If the Library as you received it specifies that a proxy can decide whether future versions of the GNU Lesser General Public License shall apply, that proxy’s public statement of acceptance of any version is permanent authorization for you to choose that version for the Library.