Jürgen Wolf

Aktuell zu C11

Grundkurs C

- Eine kompakte Einführung in die Programmiersprache C
- Vom ersten Schritt bis zum komplexen Programm
- Mit Übungen und Musterlösungen zur Lernkontrolle

2., aktualisierte Auflage

Rheinwerk Computing

Liebe Leserin, lieber Leser,

ich freue mich, dass Sie sich für dieses Taschenbuch zur C-Programmierung entschieden haben. In unserem umfangreichen Buchangebot zur Programmierung finden Sie sonst vor allem ausführliche Einsteigerliteratur und umfassende Lehr- und Handbücher. Dieses Buch haben wir für alle diejenigen gemacht, die schnell und preiswert das Basiswissen zu C erwerben wollen.

Es bietet Ihnen einen kompakten Überblick über das gesamte Grundlagenwissen der Sprache C und die Neuerungen des Standards C11 – und damit alles, was Sie brauchen, um in C zu programmieren. So ist es ideal geeignet, um sich zielgerichtet Grundkenntnisse anzueignen oder Ihr Wissen aufzufrischen.

Jedes Thema, seien es Schleifen, Funktionen, Zeiger oder komplexe Datentypen, wird in einem eigenen Kapitel dargestellt. So können Sie schnell nachschlagen, was Sie an Informationen benötigen.

Wenn Sie bisher noch nicht in C programmiert haben und die Sprache lernen wollen, gehen Sie einfach das Buch vom Anfang bis zum Ende durch. Die Kapitel führen Sie von den einfacheren Sprachelementen zu den komplexeren Strukturen und Konstrukten. Konzepte und Sprachmittel werden immer im Zusammenhang erläutert, und alles baut aufeinander auf. Am Ende jedes Kapitels gibt es Übungen und Aufgaben. Die Lösungen finden Sie im Anhang.

Übrigens: Anregungen, Verbesserungsvorschläge und Kritik sind herzlich willkommen. Ich freue mich über Ihre Rückmeldung!

Ihre Almut Poll
Lektorat Rheinwerk Computing

almut.poll@rheinwerk-verlag.de
www.rheinwerk-verlag.de
Rheinwerk Verlag · Rheinwerkallee 4 · 53227 Bonn

Hinweise zur Benutzung

Impressum

Dieses E-Book ist ein Verlagsprodukt, an dem viele mitgewirkt haben, insbesondere:

Lektorat Almut Poll, Anne Scheibe
Fachgutachten Torsten T. Will, Bielefeld
Korrektorat Isolde Kommer, Großerlach
Herstellung E-Book Denis Schaal
Covergestaltung Barbara Thoben, Köln
Satz E-Book III-satz, Husby

Wir hoffen sehr, dass Ihnen dieses Buch gefallen hat. Bitte teilen Sie uns doch Ihre Meinung mit und lesen Sie weiter auf den *Serviceseiten*.

Bibliografische Information der Deutschen Nationalbibliothek:
Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über http://dnb.d-nb.de abrufbar.

ISBN 978-3-8362-4115-1 (E-Book)
ISBN 978-3-8362-5282-9 (Bundle)

2., aktualisierte und überarbeitete Auflage 2016

www.rheinwerk-verlag.de

Inhalt

Vorwort

Mit C lernen Sie eine sehr universelle und plattformunabhängige Programmiersprache, die für fast jedes erhältliche Computersystem vorhanden ist und mit der Sie sehr ressourcensparende und schnelle Programme erstellen können. Ebenso einfach ist es, in C geschriebene Programme jederzeit auf verschiedene Systeme zu portieren, wenn man sich an den genormten Standard von C hält. Auch wenn es sich bei diesem Buch nicht um eine technische Spezifikation zu C handelt, sollten Sie sich immer der Wichtigkeit des Standards bewusst sein, wenn Sie wirklich portable Programme schreiben wollen.

Wenn Sie C-Programme schreiben und die Regeln des Sprachstandards nicht beachten, provozieren Sie außerdem ein undefiniertes Verhalten (*undefined behaviour*), das zu einem nichtportablen Programm führen kann.

Ziel dieses Buches ist es, Ihnen einen grundlegenden Einstieg in die Programmiersprache C zu vermitteln. Obgleich C im Verhältnis zu anderen Programmiersprachen einen geringen Sprachumfang hat, lassen sich damit trotzdem professionelle und plattformunabhängige Programme entwickeln. Mit diesem Buch werden Sie zwar nicht gleich zum Profi und schreiben professionelle Anwendungen. Sie erlernen jedoch die nötigen Grundlagen, auf denen Sie aufbauen können, wenn Sie die professionelle Anwendungsentwicklung betreiben – ganz gleich, ob Sie sich mit GUI-Anwendungen, Backend-Bibliotheken, Embedded Code oder etwas anderem beschäftigen.

Natürlich ist es nicht immer möglich, in einem kleinen Taschenbuch wie diesem auf alle Regeln einzugehen, und dies würde wohl gerade auf einen Einsteiger in die Sprache eher abschreckend wirken. Es könnte daher hilfreich sein, wenn Sie sich parallel zu diesem Buch – oder nach dem Buch – mit dem aktuellen Standard von C (derzeit ISO/IEC 9899:2011 oder auch C11) befassen.

Alle Beispiele im Buch lassen sich auf jeden moderneren C-Compiler verwenden. Wenn Sie einen älteren Compiler verwenden, der beispielsweise

den C89-Standard erfüllt, und im entsprechenden Abschnitt einen Hinweis finden, dass diese Funktionalität erst seit C99 oder gar C11 vorhanden ist, können Sie das Beispiel mit Ihrem älteren Compiler logischerweise nicht ausführen.

Auch ist dieses Buch keine Referenz zur Standardbibliothek und den vielen Funktionen von C. Eine solche Referenz würde weitere 300 Buchseiten füllen. Es gibt jedoch viele gute Online-Referenzen wie etwa *http://en.cppreference.com/w/c* oder *http://www.cplusplus.com/reference/clibrary/*. Außerdem enthalten die meisten Compiler eine Hilfe mit einer C-Referenz. Und nicht zu vergessen die Manpages, die mit dem Unix-Kommando `'man'` auf vielen Linux- und Unix-Systemen beheimatetet sind. Die wichtigste Referenz ist wohl die PDF-Datei des Committee Drafts zum C11-Standard unter *http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1570.pdf*.

Wenn Sie einen Blick auf den zur Drucklegung aktuellen 700-seitigen C11-Standard (ISO/IEC 9899:2011) werfen (*http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1570.pdf*), dürfte schnell klar sein, dass nach der Lektüre noch vieles übrigbleibt. Mit diesem Hintergrundwissen und dem nötigen Respekt davor, eine wirklich universelle Sprache zu lernen, können Sie sich nun an das Durcharbeiten dieses Buchs machen.

Jürgen Wolf

Kapitel 1

Einstieg in die Welt von C

Im ersten Kapitel geht es um eher theoretische Themen. Sie erfahren, wie die Sprache C aufgeteilt ist, welche Rolle die Standardbibliothek spielt und welche Schreibkonventionen in diesem Buch eingehalten werden. Auch auf die für C benötigten Werkzeuge soll kurz eingegangen werden.

1.1 Die Sprache C

Zur Drucklegung ist der C11-Standard (auch: ISO/IEC 9899:2011) aktuell. Er hat 2011 den C99-Standard abgelöst. Für den Einstieg in C ist es zunächst wichtig zu wissen, dass diese Programmiersprache aus zwei Teilen besteht:

- dem eigentlichen **Sprachkern** mit der Syntax und Semantik von C. Er umfasst im Grunde nur wenige Anweisungen und Schlüsselworte, was C zu einer sehr kleinen und schlanken Sprache macht. Im Sprachkern sind Elemente wie die eingebauten Datentypen, die Verwendung von geschweiften Klammern für Anweisungsblöcke und von runden Klammern für Funktionen, verschiedene Kontrollstrukturen wie bedingte Anweisungen oder Schleifen, Zeiger usw. enthalten.
- der **Standardbibliothek** von C mit ihren vielen externen Bibliotheksdateien, die bei Bedarf zu einem Programm hinzugefügt werden können. Somit liegen beispielsweise Ein- und Ausgabefunktionen, Dateizugriffe, Speicherverwaltung, mathematische Operationen oder die Verarbeitung von Zeichenketten in vorkompilierten Modulen vor, die für den entsprechenden Rechnertyp (Prozessor, Betriebssystem) optimiert sind.

Gerade durch diese Trennung von Sprachkern und Standardbibliothek sind C und die damit erstellten Programme so kompakt. Dadurch eignet sich C auch sehr gut für die Systemprogrammierung, weil Sie die optiona-

len Module der Standardbibliothek einfach weglassen können, wenn es keinen Sinn macht, sie zu verwenden.

Erwartungen an dieses Buch

Dieses Buch ist nicht als Kompendium für C konzipiert, sondern als **Grundkurs**. Sie lernen elementare Paradigmen von C kennen, mit dem Ziel, die Grundlagen des Sprachkernes von C verwenden zu können. Dasselbe gilt für die Standardbibliothek von C: Auch hier lernen Sie nur die grundlegenden Module kennen – für die komplette Standardbibliothek wäre wohl der doppelte Seitenumfang nötig. Auch ist dieses Buch keine technische Spezifikation zu C bzw. zur ISO/IEC 9899:2011. Wenn Sie sich tiefer in die Details von C einlesen wollen, empfehle ich Ihnen *The final draft, N1570* auf der Webseite *http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1570.pdf*.

1.2 Die C-Standardbibliothek

Auch wenn dieses Buch keine Referenz für die Standardbibliothek enthält, sollten Sie ihren Sinn kennen. So bietet der Sprachkern von C beispielsweise keinerlei Funktionen für die Ein- und Ausgabe. Hierzu werden Module von der Standardbibliothek angeboten. In welchem Umfang die Standardbibliothek vorhanden sein muss, hängt von Ihrer Umgebung ab.

Bei einer betriebssystemgestützten Implementierung, mit der es wohl die meisten Leser beim Erlernen der Sprache C zunächst zu tun haben, muss die C-Standardbibliothek in vollem Umfang vorhanden sein, wenn die Bibliothek sich als standardkonform bezeichnet. Die betriebssystemgestützte Implementierung wird auch als *hosted environment* bezeichnet.

Wenn Sie C im Embedded-Bereich verwenden, haben Sie es in der Regel mit einer freistehenden Implementierung zu tun. Diese Form der Umgebung muss nur einen bestimmten Teil der Standardbibliothek anbieten, damit sie als standardkonform bezeichnet werden kann. Die freistehende Implementierung bzw. freistehende Umgebung wird auch als *freestanding environment* bezeichnet.

Headerdateien und Programmbibliothek

Der Aufbau der Standardbibliothek bleibt dem Entwickler häufig verborgen. In den kommenden Programmen werden Sie bei jedem Beispiel sogenannte *Headerdateien* mit der Dateiendung **.h* hinzufügen. In diesen Header-Dateien werden unter anderem fertige Funktionen und Typen deklariert, die im Programmcode verwendet werden können, wenn die Headerdatei eingebunden wurde.

Der tatsächliche Inhalt bzw. die Implementation der Funktionen ist dann in die *Programmbibliothek* ausgelagert.

Bei Headerdateien und Programmbibliothek handelt es sich also um zwei unterschiedliche Dinge: Meistens sind Headerdateien separate einzelne Dateien, deren Namensgebung vom Standard genormt wird. Die Implementierung der Programmbibliothek hingegen wird meistens vom Compilerhersteller verwaltet und kann daher variieren.

Übersicht zu den Headerdateien

In Tabelle 1.1 finden Sie einen Überblick zu den verschiedenen Headerdateien, die Ihnen heute mit C11 zur Verfügung stehen. Da es wohl immer noch Compiler gibt, welche den C11-Standard nur teilweise bis gar nicht implementiert haben, sehen Sie auch, ab welchem Standard die entsprechende Headerdatei der Standardbibliothek vorhanden ist.

Headerdatei	Standard	Bedeutung
`<assert.h>`	C89/C90	Assertions; Fehlersuche
`<complex.h>`	C99	Komplexe Zahlenarithmetik
`<ctype.h>`	C89/C90	Test auf bestimmte Zeichentypen
`<errno.h>`	C89/C90	Makros mit Fehlercodes
`<fenv.h`	C99	Einstellungen für die Gleitkommaberechnungen
`<float.h>`	C89/C90	Limits für Gleitkommazahlen

Tabelle 1.1 C-Standardbibliothek-Headerdateien

Headerdatei	Standard	Bedeutung
<inttypes.h>	C99	Konvertierungsfunktionen für Ganzzahltypen
<iso646.h>	C95/NA1	Alternative Schreibweise für logische und bitweise Operatoren.
<limits.h>	C89/C90	Größe eingebauter Typen
<locale.h>	C89/C90	Einstellungen des Gebietsschemas
<math.h>	C89/C90	Mathematische Funktionen
<setjmp.h>	C89/C90	Nichtlokale Sprünge
<signal.h>	C89/C90	Signalverarbeitung
<stdalign.h>	C11	Makros für Speicherausrichtung
<stdarg.h>	C89/C90	Variable Anzahl von Argumenten
<stdatomic.h>	C11	Typen für atomare Operationen für Threads
<stdbool.h>	C99	Boolesche Variablen
<stddef.h>	C89/C90	Zusätzliche Typendefinitionen
<stdint.h>	C99	Ganzzahltypen mit fester Breite
<stdio.h>	C89/C90	Ein-/Ausgabe
<stdlib.h>	C89/C90	Allgemeine Standardfunktionen
<stdnoreturn.h>	C11	Definition des Noreturn-Makros
<string.h>	C89/C90	Funktionen für Zeichenketten
<tgmath.h>	C99	Typgenerische Makros für mathematische Funktionen

Tabelle 1.1 C-Standardbibliothek-Headerdateien (Forts.)

Headerdatei	Standard	Bedeutung
`<threads.h>`	C11	Unterstützung von Multithreads
`<time.h>`	C89/C90	Datum und Uhrzeit
`<uchar.h>`	C11	Unterstützung von Unicode-Zeichen (UTF-16- und UTF-32-kodiert)
`<wchar.h>`	C95/NA1	Unterstützung für Unicode-Zeichen
`<wctype.h>`	C95/NA1	Wie `<ctype.h>`, nur für Unicode

Tabelle 1.1 C-Standardbibliothek-Headerdateien (Forts.)

1.3 Die nötigen Werkzeuge für C

Um aus einem einfachen C-Quellcode eine ausführbare Datei zu erstellen, gibt es im Grunde zwei Wege – einen ungemütlichen und einen gemütlichen. Der unbequeme Weg (wobei dies auch Ansichtssache ist) lautet:

1. Den Quellcode in einen beliebigen *ASCII-Texteditor* eintippen und abspeichern.
2. Den Quellcode mit einem *Compiler* übersetzen, wodurch eine Objektdatei (**.obj* oder **.o*) erzeugt wird.
3. Die Objektdatei mit einem *Linker* binden, was eine ausführbare Datei erzeugt. Der Linker sucht dabei alle benötigten Funktionen aus den Standardbibliotheken heraus und fügt sie anschließend dem fertigen Programm hinzu.

Für diesen manuellen Weg bräuchten Sie also nur einen einfachen ASCII-Texteditor, einen Compiler und einen Linker. Den Compiler und den Linker müssten Sie in der Kommandozeile aufrufen. In der Praxis wird allerdings eher selten zwischen Compiler und Linker unterschieden, und wenn die Rede von einem Compiler ist, ist damit meistens gleichzeitig auch der Linker als komplette Einheit gemeint. Die Abbildung 1.1 zeigt diesen Übersetzungsvorgang in vereinfachter Form.

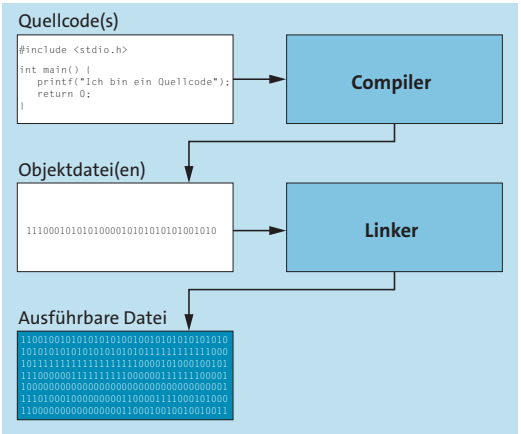

Abbildung 1.1 Die Grundlagen, um aus einem Quellcode mit einfachsten Mitteln eine ausführbare Datei zu generieren

Der bequemere Weg führt über eine *All-in-one*-Lösung, eine *Entwicklungsumgebung*. Wie der Name schon vermuten lässt, finden Sie hier alles, was Sie zum Programmieren benötigen, in einem Fenster.

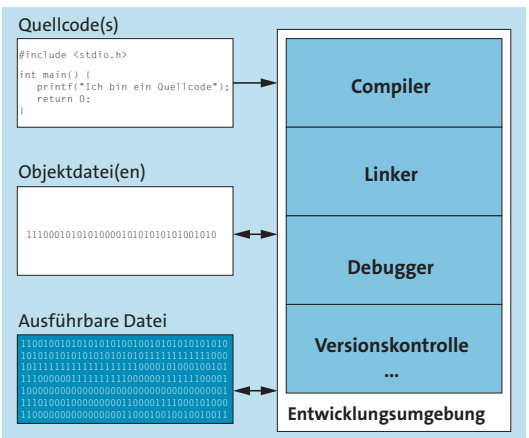

Abbildung 1.2 Der komfortablere Weg, ein ausführbares Programm zu erstellen, führt über eine Entwicklungsumgebung.

Eine Entwicklungsumgebung bietet natürlich meistens weitaus mehr als nur Texteditor, Compiler und Linker. Häufig finden Sie hier auch einen Debugger zum schrittweisen Durchlaufen einer Anwendung, eine Projektverwaltung, um die Übersicht zu behalten, oder einen Profiler, um die Laufzeit des Programms zu analysieren.

Ich empfehle Ihnen, sich einen aktuellen Compiler zu besorgen und zu verwenden. Wenn es Ihnen darauf ankommt, dass er auch mit dem neueren C11-Standard (zum Teil) etwas anfangen kann, bieten sich die folgenden drei Compiler an:

- **gcc aus GNU GCC**: Der C-Compiler gcc aus der GNU Compiler Collection (kurz: GCC) wird auf vielen Plattformen verwendet und ist sehr beliebt, um neue Features auszuprobieren. Unter Linux ist er gewöhnlich die erste Wahl. Unter Windows steht die Portierung MinGW (Minimal GNU for Windows) zur Verfügung.
- **Clang der LLVM**: Clang aus LLVM ist der C-Standardcompiler für die Mac-OS-Entwicklung schlechthin, mit einer sehr guten Unterstützung für C11. Aber auch für Linux können Sie diesen Compiler kostenlos nachinstallieren.
- **Pelles C**: Diese schlanke Entwicklungsumgebung für Microsoft Windows verwendet eine modifizierte und erweiterte Version des LCC-Compilers mit Unterstützung für C99 und C11 und dürfte somit die erste Wahl für die reine Programmierung in C unter Windows sein.

1.4 Übersetzen mit der Entwicklungsumgebung

Sobald Sie Ihre Entwicklungsumgebung heruntergeladen und installiert haben, können Sie ein erstes Projekt anlegen und aus einem einfachen Quelltext ein ausführbares Programm erstellen. Ich zeige Ihnen beides anhand von Pelles C, aber der Vorgang ist in der Regel bei anderen Entwicklungsumgebungen vergleichbar – nur lautet hier und da ein Befehl anders oder findet sich an einer anderen Stelle.

1. Am einfachsten ist es, wenn Sie ein neues Projekt anlegen, weil sich dann die Entwicklungsumgebung gleich um viele kleinere Details kümmert und Sie sich leichter tun, wenn Sie später weitere Dateien

zum Projekt hinzufügen wollen. Ein neues Projekt starten Sie mit DATEI • NEU • PROJEKT.

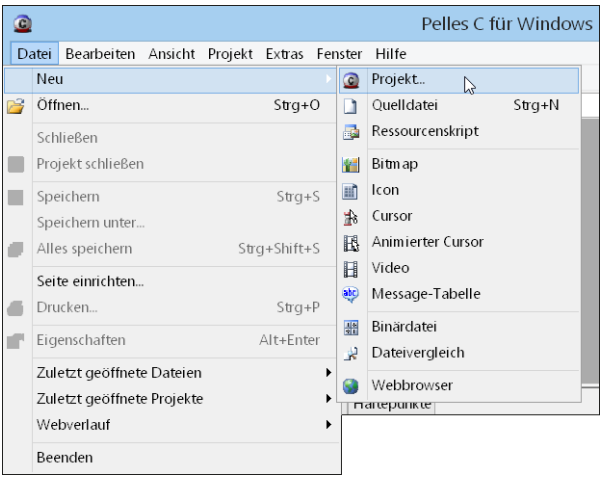

Abbildung 1.3 Zunächst sollten Sie ein neues Projekt anlegen.

2. Gewöhnlich folgt jetzt ein Dialog mit der Abfrage, welche Art von Projekt Sie erstellen wollen, wie sein Name lauten und wo es gespeichert werden soll. Häufig ist es am einfachsten, ein leeres C-Projekt oder (wie hier) eine Konsolenanwendung (hier: Win64-Konsolenanwendung (EXE)) zu erzeugen. Wird kein passendes Projekt als Vorlage aufgelistet, bieten Entwicklungsumgebungen häufig auch an, entsprechende Vorlagen herunterzuladen.
3. Abhängig von der Entwicklungsumgebung finden Sie vielleicht schon ein C-Grundgerüst mit Quellcode vor. Bei anderen Entwicklungsumgebungen müssen Sie erst noch ein neues Element bzw. eine neue Datei hinzufügen. Neue Elemente bzw. Dateien werden Sie auch hinzufügen müssen, wenn Sie den Quellcode in mehrere Module aufteilen. Hierbei ist immer darauf zu achten, dass Sie die neue Datei zum Projekt hinzufügen und nicht einfach nur eine neue, leere Datei anlegen.

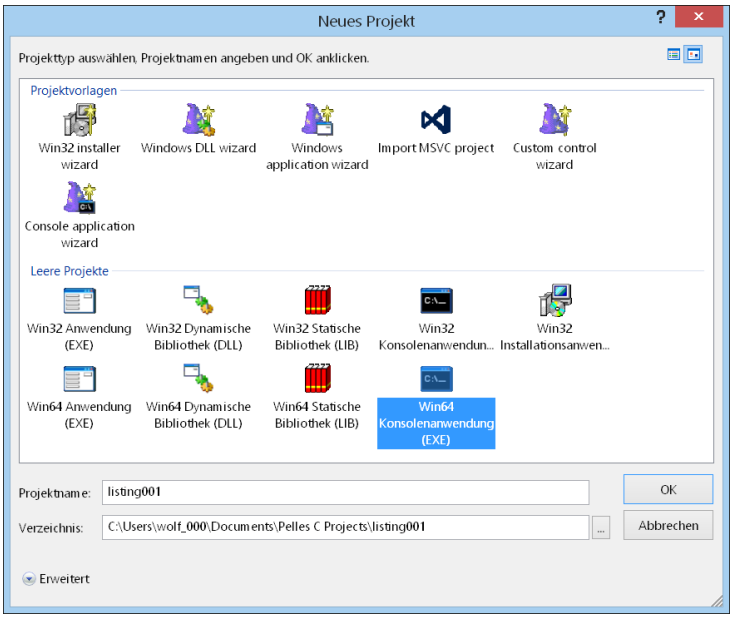

Abbildung 1.4 Normalerweise hilft Ihnen ein Assistent dabei, das Projekt zu erstellen.

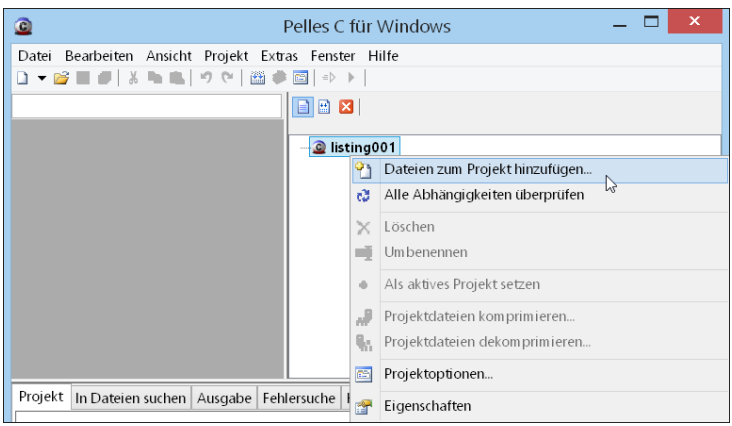

Abbildung 1.5 Eine neue Datei zum Projekt hinzufügen

4. Häufig hilft Ihnen auch hier ein weiterer Dialog, eine Datei zum Projekt hinzuzufügen. Bei Ihren ersten Beispielen werden Sie ohnehin eine Quelldatei mit der Endung `*.c` hinzufügen wollen. Bei späteren Projekten werden Sie des Öfteren neben einer C-Datei auch eine eigene Headerdatei mit der Endung `*.h` hinzufügen. Bei Pelles C müssen Sie nur den Dateinamen (hier: *listing001.c*) eintippen und auf die Schaltfläche ÖFFNEN klicken.

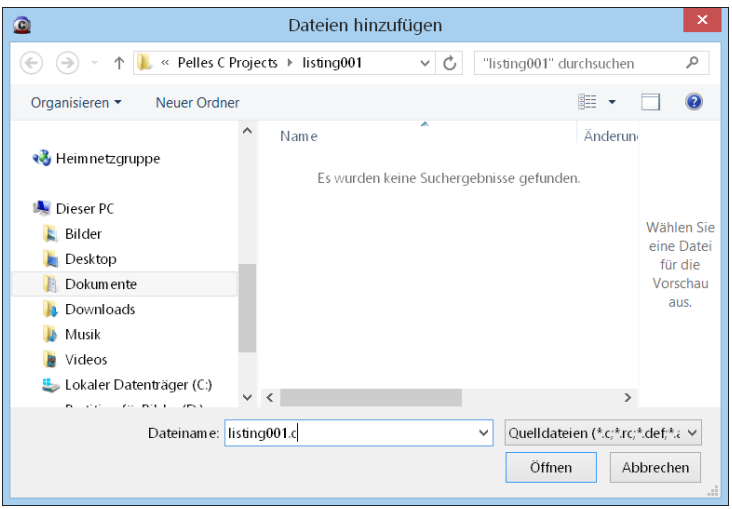

Abbildung 1.6 Den Dateinamen für die neue Projektdatei eingeben

5. Sobald Sie die Datei zum Projekt hinzugefügt haben, können Sie anfangen, den Quelltext einzutippen. Nach dem Abspeichern des Quellcodes können Sie diesen übersetzen (kompilieren) und ausführen. Bei Pelles C können Sie den Quelltext über PROJEKT • KOMPILIERE LISTING001.C nur kompilieren, über PROJEKT • ERZEUGE LISTING001.EXE kompilieren, linken und ein ausführbares Programm daraus machen oder ihn mit PROJEKT • LISTING001.EXE AUSFÜHREN kompilieren, linken und auch gleich ausführen.

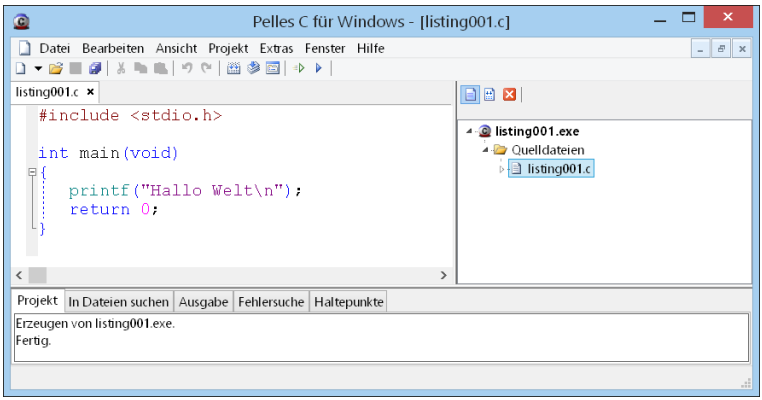

Abbildung 1.7 Quelltext eintippen, dann das Programm übersetzen und starten

1.5 Übersetzen mit gcc und clang

Die Verwendung von Kommandozeilen-Compilern wie `gcc` oder `clang` soll hier auch kurz beschrieben werden. Hierbei schreiben Sie Ihren Quelltext mit einem beliebigen ASCII-Editor und speichern ihn mit der Endung **.cpp* ab. In der Kommandozeile übersetzen Sie den Quelltext dann mit `gcc` oder `clang`. Gerade bei kleineren Listings wie in diesem Buch ist dieser Vorgang weniger aufwendig, als den Quelltext mit einer Entwicklungsumgebung zu übersetzen.

Dabei gehe ich davon aus, dass Sie die Kommandozeile kennen und sich mit unterschiedlichen Kommandos durch die Verzeichnisse bewegen können. Wenn Sie den Quelltext geschrieben und abspeichert haben, öffnen Sie die Kommandozeile, und wechseln Sie in das Verzeichnis, in dem Sie den Quelltext gespeichert haben. Tippen Sie Folgendes ein (*listing001.c* sei der abgespeicherte C-Quelltext):

```
$ gcc -o listing001 listing001.c
$ ./listing001
...
```

Mit dem Schalter `-o` wird hier veranlasst, dass der Compiler aus der Quelldatei *listing001.c* die ausführbare Datei *listing001* macht. Sie können statt *listing001* natürlich auch einen anderen Programmnamen verwenden. In der folgenden Zeile wird daraufhin die ausführbare Datei in der Kommandozeile mit dem Programmnamen gestartet.

In der Praxis empfiehlt es sich aber, mehrere solcher Flags wie `-o` zu verwenden, um dem Compiler bei einer Warnung oder Fehlern mehr Informationen zu entlocken. Meine minimale Zeile zum Übersetzen eines Quelltextes lautet daher:

```
$ gcc -Wall -o listing001 listing001.c
$ ./listing001
...
```

Der Schalter `-Wall` gibt mir alle sinnvollen Warnungen des Compilers aus. *Wall* setzt sich aus ***W**arning* und ***all*** zusammen. Sollten Sie den Quellcode nur kompilieren (und nicht zusätzlich linken) wollen, müssen Sie statt `-o` den Schalter `-c` verwenden.

Schalter für C99 und C11

Wenn der Compiler nicht schon von Haus aus den C11-Standard verwendet, können Sie diesen mit dem Flag `-std=c11` aktivieren. Dasselbe gilt auch für den vorherigen C99-Standard, den Sie mit `-std=c99` aktivieren können – beispielsweise:

```
$ gcc -Wall -o prgname prgname.c -std=c11
$ ./prgname
...
```

1.6 Listings zum Buch

Zwar ist es empfehlenswert, dass Sie die Beispiele im Buch selbst eintippen, aber aus eigener Erfahrung weiß ich, dass bei dem einen oder anderen Beispiel einfach die Lust oder Zeit dazu fehlt. Außerdem verzichte ich an manchen Stellen auf seitenlangen Code und zeige nur die zum Thema

passenden Codezeilen. Daher können Sie die (kompletten) Listings aus dem Buch sowie die Musterlösungen zu den Aufgaben von der Webseite *https://www.rheinwerk-verlag.de/4108* herunterladen.

Alle Listings wurden auf C99 und C11 getestet. Dinge, die im C11-Standard neu hinzugefügt wurden, funktionieren logischerweise in C99 nicht. Für die Tests wurden die zur Drucklegung aktuellsten Compiler wie gcc, clang und der Pelles C (Version 8.0) zugrundeliegende Compiler verwendet.

Niemand ist zu hundert Prozent perfekt. Sollten Sie Fehler in einem Listing finden, würde ich mich über ein kurzes Feedback sehr freuen.

1.7 Kontrollfragen und Aufgaben im Buch

Am Ende jedes Kapitels finden Sie einige Kontrollfragen und Aufgaben. Mit diesen Tests können Sie überprüfen, ob Sie das Gelesene grundsätzlich verstanden haben.

1.8 Aufgabe

Bevor Sie mit dem Buch fortfahren, haben Sie eine sehr wichtige Aufgabe vor sich: Tippen Sie den nachfolgend abgedruckten C-Quellcode in eine Entwicklungsumgebung oder einen ASCII-Editor ihrer Wahl.

```
#include <stdio.h>

int main(void) {
   printf("Hallo Welt\n");
   return 0;
}
```

Übersetzen Sie dann das Programm, und bringen Sie es zur Ausführung. Lesen Sie erst weiter, wenn Sie das Programm eingetippt, gespeichert und zur Ausführung gebracht haben.

Kapitel 2

Erste Schritte in C

In diesem Kapitel unternehmen Sie die ersten Schritte in C. Sie schreiben Ihr erstes Programm und erfahren, was alles zu seinem Grundgerüst gehört. Weiterhin lernen Sie schon einmal die Möglichkeiten kennen, in C etwas auf dem Bildschirm auszugeben.

2.1 Das erste Programm in C

Ihr erstes Programm wird einfach einen Text auf dem Bildschirm ausgeben. Tippen Sie in den Texteditor Ihrer Wahl (oder in einer Entwicklungsumgebung) folgenden Quellcode:

```
00 // kap002/listing001.c
01 #include <stdio.h>

02 int main(void) {
03    printf("Mein erstes Programm\n");
04    return 0;
05 }
```

Nachdem Sie das Beispiel *listing001.c* in eine ausführbare Datei übersetzt haben, gibt das Programm die Textfolge »Mein erstes Programm« aus. Zugegeben, noch nicht sehr beeindruckend – aber hier haben Sie schon ein komplettes Grundgerüst für ein C-Programm vor sich. In den folgenden Absätzen analysieren wir dieses erste Programm ein wenig.

Die Zeile **(00)** ist lediglich ein Kommentar und wird beim Übersetzen des Programms verworfen. In der Zeile **(01)** finden Sie mit `#include` einen Befehl für den Präprozessor. Der Präprozessor ist wiederum ein Teil des Compilers – oder genauer: ein Programm, das vor dem Compiler ausgeführt wird. Damit bestimmen Sie, dass der Präprozessor die nötigen Bib-

liothekquellcodes für unser Listing aus der dahinter angegebenen Datei `stdio.h` einkopieren soll. Entfernen Sie diese Zeile, bekommen Sie eine Fehlermeldung, da die Funktion `printf` einen *Prototyp* benötigt. Diese Funktion wird in der Zeile **(03)** verwendet. Der Compiler kann also mit der Funktion `printf` nichts anfangen, wenn die Headerdatei `stdio.h` nicht mit einkopiert wurde, in der sich dieser Prototyp befindet.

In der Zeile **(02)** haben Sie mit `main()` die Hauptfunktion des Programms. Jedes fertige und ausführbare Programm benötigt eine solche `main`-Funktion. Diese ist der Einsprungspunkt (engl. *program startup*). Wenn Sie ein Programm starten, werden von dort aus gewöhnlich weitere Funktionen aufgerufen. Ohne eine `main`-Funktion kann der Linker später kein ausführbares Programm erstellen. Das `void` innerhalb der runden Klammern der `main`-Funktion steht für einen leeren Datentyp; die Funktion hat also keine Parameter. Aber dazu später mehr.

Der Anfang und das Ende der `main`-Funktion – und von Funktionen überhaupt – wird zwischen geschweifte Klammern `{ ... }` gesetzt, die hier in der Zeile **(02)** hinter der `main`-Funktion geöffnet und in der Zeile **(05)** geschlossen werden. Alles zwischen diesen geschweiften Klammern wird als *Anweisungsblock* bezeichnet. Die Klammern fassen somit mehrere Anweisungen, welche die `main`-Funktion auszuführen hat, zu einem Block zusammen. Beachten Sie, dass die geschweiften Klammern nicht unbedingt so positioniert werden müssen wie in meinem Beispiel. Das Beispiel entspricht einfach meinem Schreibstil. Sie könnten die geschweiften Klammern auch folgendermaßen setzen:

```
{ /* Anweisungen */ }
```

Oder so:

```
{
   /* Anweisungen */
}
```

Im Anweisungsblock wird mit der Zeile **(03)** die Funktion `printf` ausgeführt. Wie bereits erwähnt, ist diese Funktion in der Headerdatei `stdio.h`

deklariert und wird für die formatierte Ausgabe von Text verwendet. Der auszugebende Text muss dabei immer zwischen die doppelten Hochkommata (`"Mein erstes Programm\n"`) gestellt werden.

In der Zeile **(04)** geben Sie der Hauptfunktion `main` mit `return` den Rückgabewert `0` zurück. Im Normalfall bedeutet dies, dass das Programm sauber beendet wurde.

Am Ende der Zeilen **(03)** und **(04)** steht jeweils ein Semikolon (Strichpunkt). Mit diesem Zeichen wird in C eine Anweisung beendet. Mit der Programmausführung geht es dann in der nächsten Zeile weiter. So wird zum Beispiel mit der nächsten Zeile fortgefahren und die Anweisung `return 0;` ausgeführt, wenn die Anweisung `printf(...);` fertig ausgeführt ist.

2.2 Die Funktion printf

Da Sie `printf` in den folgenden Beispielen noch häufig verwenden werden, soll diese Funktion hier etwas genauer behandelt werden. Aus dem vorherigen Abschnitt wissen Sie bereits, dass `printf` dazu verwendet wird, eine Zeichenkette (engl. *string*) formatiert auf dem Bildschirm auszugeben. Die Zeichenkette steht dabei immer zwischen zwei doppelten Hochkommata.

Die Headerdatei stdio.h

Die Headerdatei `stdio.h` müssen Sie im Grunde bei fast jedem C-Programm hinzufügen. Dort sind alle Standardfunktionen für die Standard-Ein-/Ausgabe deklariert (stdio steht für **St**andar**d** **I**nput **O**utput).

Wollen Sie eine Zeichenkette über das Zeilenende fortsetzen, müssen Sie das Backslash-Zeichen (\) an das Ende der Zeile setzen, beispielsweise:

```
// Ausgabe über mehrere Zeilen
printf("Ich werde ein \
Filmstar\n");
```

Beachten Sie, dass alle Leerzeichen nach dem Backslash in der nächsten Zeile ebenfalls bei der Ausgabe berücksichtigt werden. Das String-Literal

von `printf` muss allerdings nicht zwangsläufig nur aus einem konstanten Text zwischen den Hochkommata bestehen.

Ein Beispiel hierzu:

```
printf("Gib mir zehn: %d", 10);
```

In diesem Beispiel dient `%d` als Platzhalter für einen Zahlenwert und gibt zugleich an, dass dieser als Dezimalzahl dargestellt werden soll.

Es kann mehrere solche Platzhalter in einer Zeichenkette geben; die Werte werden dann in der angegebenen Reihenfolge eingesetzt. Für einen Wert kann auch ein längerer Ausdruck stehen, z. B. `2+3`. Ein Beispiel:

```
printf("Gib mir zehn: %d - gib mir fünf: %d", 10, 2+3);
```

Jetzt werden mehrere Platzhalter verwendet. Diese werden in C auch als *Formatanweisungen* bezeichnet. Alle Formatanweisungen beginnen mit einem `%`-Zeichen. Dahinter folgt ein Buchstabe, der den Datentyp des Formats angibt. `%d` steht z. B. für eine dezimale Ganzzahl, wie Sie bereits erfahren haben. Die Formatanweisung lässt sich noch erweitert formatieren.

Formatanweisungen

Neben `%d` für die Dezimale gibt es noch viele andere Kürzel zur Steuerung der formatierten Ausgabe verschiedener Daten. Diese sogenannten Formatanweisungen lernen Sie in Abschnitt 14.7, »Funktionen zur formatierten Ein-/Ausgabe«, kennen.

2.3 Zeichensätze in C

In C wird zwischen dem Basis-Ausführungszeichensatz und den Escape-Sequenzen unterschieden. Der Basis-Ausführungszeichensatz beinhaltet alle Zeichen, die für das Schreiben von Programmen verwendet werden können. Die Zeichen von Escape-Sequenzen hingegen werden erst bei der Ausführung des Programms interpretiert.

2.3.1 Basis-Ausführungszeichensatz

Sie können in Ihren C-Programmen die folgenden Zeichen verwenden:

- Dezimalziffern:
  `1 2 3 4 5 6 7 8 9 0`
- Buchstaben des englischen Alphabets:
  `A B C D E F G H I J K L M N O P Q R S T U V W X Y Z`
  `a b c d e f g h i j k l m n o p q r s t u v w x y z`
- Grafiksymbole:
  `! " % & / ( ) [ ] { } \ ? =`
  `' # + * ~ - _ . : ; , | < > ^`
- Whitespace-Zeichen wie Leerzeichen, Tabulatorzeichen und Zeilenumbruch

2.3.2 Escape-Sequenzen

Zum Zeichensatz von C gehören auch die sogenannten Escape-Sequenzen. Dies sind nicht druckbare Zeichen innerhalb eines String-Literals. Sie werden auch Steuerzeichen genannt, weil sie die Ausgabe des Strings zusätzlich steuern, z. B. die Cursorposition ändern oder einen Ton hinzufügen können. Escape-Sequenzen bestehen immer aus einem Backslash und einem weiteren Zeichen (siehe Tabelle 2.1).

Escape-Sequenz	Bedeutung
`\a`	Akustisches Warnsignal (*beep*).
`\b`	Setzt den Cursor um eine Position nach links (*backspace*).
`\f`	Löst einen Seitenvorschub aus. Macht Sinn bei Programmen, die etwas ausdrucken (*formfeed*).
`\n`	Setzt den Cursor an den Anfang der nächsten Zeile (*newline*).

Tabelle 2.1 Steuerzeichen in Zeichenkonstanten

Escape-Sequenz	Bedeutung
\r	Setzt den Cursor an den Anfang der aktuellen Zeile (*carriage return*).
\t	Führt einen Zeilenvorschub aus und setzt den Cursor an die nächste horizontale Tabulatorposition – meistens acht Leerzeichen weiter (*horizontal tab*).
\v	Setzt den Cursor auf die nächste vertikale Tabulatorposition (*vertical tab*).
\"	Gibt das Zeichen " aus.
\'	Gibt das Zeichen ' aus.
\?	Gibt das Zeichen ? aus.
\\	Gibt das Zeichen \ aus.
\nn \Onn	Gibt einen Oktalwert aus (beispielsweise \033 = Escape-Zeichen). n ist eine Zahl zwischen 0 und 7.
\xhh	Gibt einen Hexadezimalwert aus (h = 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F).

Tabelle 2.1 Steuerzeichen in Zeichenkonstanten (Forts.)

Das folgende Listing demonstriert die Verwendung der Steuerzeichen in einem C-Programm:

```
00  // kap002/listing002.c
01  #include <stdio.h>

02  int main(void) {
03     printf("Ich bin ein \"Blindtext\"\n");
04     printf("\tNoch mehr Text\n");
05     printf("Ich werde ueberschrieben\r");
06     printf("Von mir, einem weiteren Blindtext\n");
```

```
07      printf("Das Leer \bzeichen wird entfernt\n");
08      return 0;
09  }
```

Die Ausgabe des Programms lautet:

```
Ich bin ein "Blindtext"
        Noch mehr Text
Von mir, einem weiteren Blindtext
Das Leerzeichen wird entfernt
```

Die Ausgabe spricht eigentlich für sich, aber trotzdem will ich das Beispiel ein wenig kommentieren. In der Zeile **(03)** wird der Cursor mit `\n` in die nächste Zeile und das Wort *Blindtext* zwischen zwei Hochkommata gesetzt. Zeile **(04)** löst zuerst einen Tabulatorvorschub mit `\t` aus. Zeile **(05)** wird zwar ausgegeben, aber am Ende setzen wir den Cursor mit `\r` an den Anfang der Zeile. Daher wird die Ausgabe der Zeile **(05)** von der Ausgabe der Zeile **(06)** überschrieben. In der Zeile **(07)** wird der Cursor zwischen einem Leerzeichen mit `\b` um eine Position nach links geschoben.

2.4 Symbole von C

Jede Programmiersprache hat einen festgelegten Satz gültiger Symbole, und es ist daher unerlässlich, dass Sie die Regeln für gültige Symbole von C kennen.

2.4.1 Bezeichner

Bezeichner sind Namen für Objekte in einem Programm, die vom Programmierer festgelegt werden, etwa Variablen, Funktionen usw. Für einen gültigen Bezeichner gelten folgende Regeln:

- Namen bestehen aus Buchstaben aus dem Basis-Ausführungszeichensatz, Ziffern und Unterstrichen.
- Das erste Zeichen darf keine Zahl sein. Ein Bezeichner darf also mit einem Buchstaben oder einem Unterstrich beginnen.

- Es wird zwischen Groß- und Kleinbuchstaben unterschieden (engl.: *case sensitive*). Somit sind »fun«, »Fun« und »FUN« drei verschiedene Bezeichner.
- Als Bezeichner darf kein reserviertes Schlüsselwort von C verwendet werden. Die Schlüsselwörter von C finden Sie in Abschnitt 2.4.2, »Reservierte Schlüsselwörter«.

Zur Demonstration sollen einige falsche und richtige Bezeichner aus einem Codeausschnitt gezeigt werden. In der Codezeile

```
int maxSpieler = 100;
```

ist `maxSpieler` der einzige Bezeichner, den Sie als Programmierer festlegen, und zwar der Name einer Variablen. `100` ist ein Literal für die Zahl Hundert. Der Variablen `maxSpieler` wird der Wert 100 zugewiesen. `int` ist ein Schlüsselwort von C und gibt an, dass man der Variablen `maxSpieler` ganze Zahlen zuweisen kann.

Weitere Beispiele für gültige – und unterschiedliche – Bezeichner sind:

```
maxspieler, ival, i
```

Ungültig sind zum Beispiel:

```
2terSpieler, zweiter Spieler, for, _Bool, Spieler(maxAnz)
```

Die Gründe: Eine Ziffer am Anfang wie bei `2terSpieler` ist nicht erlaubt, Leerzeichen wie bei `zweiter Spieler` sind nicht erlaubt, `for` ist ein Schlüsselwort, `_Bool` ebenso, und die Klammern bei `Spieler(maxAnz)` sind nicht erlaubt.

Nicht verwendet werden sollten zum Beispiel:

```
_Spieler, __spieler, _i3
```

Diese Bezeichner mit einem führenden Unterstrich liegen im reservierten Bereich, sind also *reservierte Bezeichner*.

Reservierte Bezeichner

Auf Bezeichner, die mit zwei sequenziellen Unterstrichen oder einem Unterstrich, gefolgt von einem Großbuchstaben beginnen, sollten Sie

verzichten, weil sie für C-Implementierungen reserviert sind. Bezeichner wie `__asdf` sind gewöhnlich für Compilerzwecke, Bezeichner wie `_Asdf` für Betriebssystem- und Bibliothekszwecke gedacht.

2.4.2 Reservierte Schlüsselwörter

Reservierte Schlüsselwörter haben als Bestandteile der Sprache C eine festgelegte Bedeutung. Sie dürfen nicht anderweitig verwendet werden. So dürfen Sie beispielsweise keine Variable mit dem Bezeichner `int` verwenden, da es auch einen Basisdatentyp mit diesem Namen gibt. Der Compiler würde sich ohnehin darüber beschweren. In der Tabelle 2.2 finden Sie die reservierten Schlüsselwörter in C.

auto	break	case	char
const	continue	default	do
double	else	enum	extern
float	for	goto	if
int	long	register	return
short	signed	sizeof	unsigned
void	volatile	while	
C99			
inline	restrict	_Bool	_Complex
_Imaginary			
C11			
_Alignas	_Alignof	_Atomic	_Generic
_Noreturn	_Static_assert	_Thread_local	

Tabelle 2.2 Reservierte Schlüsselwörter in C

Vielleicht ist Ihnen aufgefallen, dass `printf` kein Schlüsselwort und nach den Regeln ein erlaubter Bezeichner ist. Aber `printf` hat doch eine festgelegte Bedeutung? `printf` ist in der Tat ein erlaubter Bezeichner – er wurde nur bereits verwendet, nämlich von den Programmierern, deren Code Sie über `stdio.h` einbinden. Es ist der Name der Funktion, die Sie für die Ausgabe kennen. Ihre eigene Funktion sollten Sie vernünftigerweise nicht genauso nennen, wenn Sie `stdio.h` verwenden. Bei einer Kollision dieser Art wird sich der Compiler bei Ihnen beschweren. Vom Grundprinzip von C her ist es nicht verboten, `printf` als Bezeichner zu verwenden, weil Sie theoretisch auch Ihr eigenes `printf` programmieren und dieses dann anstelle der Version der Standardbibliothek verwenden könnten. Aus der Einführung wissen Sie ja noch, dass C aus einem Sprachkern und der Standardbibliothek besteht. Es ist also lediglich verboten, Bezeichner des Sprachkerns zu verwenden.

2.4.3 Literale

Als Literale werden Zahlen, Zeichenketten und Wahrheitswerte im Quelltext bezeichnet. Sie müssen ebenfalls nach einem bestimmten Muster aufgebaut sein. Literale sind von einer Programmiersprache definierte Zeichenfolgen für die Darstellung von Werten, z. B. `10` für die Zahl Zehn oder `"Auto"` für das Wort Auto.

Zahlenliterale

Vielleicht überrascht es Sie: Literale für Zahlenwerte können nicht nur Ziffern enthalten, sondern auch Buchstaben. Warum? Nun, es gibt ja nicht nur das Dezimalsystem, und C versteht auch Zahlen, die Sie oktal, hexadezimal oder binär angeben. Für Hexadezimalzahlen werden die Buchstaben A bis F gebraucht, um überhaupt die nötigen 16 verschiedenen Zeichen zusammenzubekommen (a bis f sind ebenfalls erlaubt). Außerdem werden der Buchstabe x und die Ziffer 0 benötigt, um anzuzeigen, dass ein Zahlenwert anders als dezimal zu verstehen ist. Das funktioniert über ein Präfix, also eine vorangestellte Zeichenfolge:

- Literale für Hexadezimalzahlen beginnen mit dem Präfix 0x oder 0X.
- Literale für Oktalzahlen beginnen mit der Ziffer 0 als Präfix.
- Literale für Dezimalzahlen sehen aus wie gewohnt, dürfen aber nicht mit der Ziffer 0 beginnen.

Darüber hinaus werden Buchstaben benötigt, um den Datentypen genauer zu bestimmen, und zwar über Suffixe, also angehängte Zeichenfolgen. So stehen `U` oder `u` für eine positive Zahl ohne Vorzeichen (für engl. *unsigned*, vorzeichenlos), `L` oder `l` für den Datentyp `long` für ganze Zahlen, die größer sein dürfen als solche ohne das `L`. `F` oder `f` stehen für eine Fließkommazahl.

Und schließlich verwenden Dezimalzahlen einen Punkt (.), wo im Deutschen ein Komma steht, also vor den »Nachkommastellen«.

Einige Beispiele:

- 20 – die Zahl 20
- 020 – die Zahl 16, oktal dargestellt
- 0x20 – die Zahl 32, hexadezimal dargestellt
- 0X1a – die Zahl 26, hexadezimal dargestellt
- 40000L – die Zahl 40.000, und zwar für den Datentyp `long` (siehe auch dazu Abschnitt 3.4)

Zeichenliterale

Ein Zeichenliteral wird zwischen einfachen Hochkommata (*single quotes*) eingeschlossen (`'A'`, `'B'`, `'C'`, ... `'$'`, `'&'` usw.). Wenn Sie nicht druckbare Zeichen (siehe Abschnitt 2.3.2, »Escape-Sequenzen«) wie beispielsweise einen Tabulator oder Zeilenvorschub darstellen wollen, müssen Sie eine Escape-Sequenz (auch Steuerzeichen oder Fluchtsequenz genannt) verwenden. Escape-Sequenzen werden mit einem Backslash (\) eingeleitet (z. B. ein Tabulator = `'\t'` oder ein Zeilenvorschub = `'\n'`).

Zeichenkettenliteral

Eine Zeichenkette (häufig auch String genannt) ist eine Sequenz von Zeichen, die zwischen doppelte Hochkommata (*double quotes*) gestellt werden (beispielsweise `"Ich bin eine Zeichenkette"`). Es ist sehr wichtig zu wissen, dass jede Zeichenkette um ein Zeichen länger ist als (sichtbar) dargestellt. Gewöhnlich werden Zeichenketten durch das Zeichen mit dem ASCII-Wert 0 (nicht die dezimale Null) abgeschlossen (0x00 oder als einzelnes Zeichen `'\0'`). Diese ASCII-0 kennzeichnet immer das Ende einer

Zeichenkette. Somit enthält beispielsweise die Zeichenkette `"ABC"` vier Zeichen, weil am Ende auch das Zeichen 0x00 (oder `'\0'`) abgelegt ist.

2.4.4 Einfache Begrenzer

Um einzelne Symbole voneinander zu trennen, werden sogenannte Begrenzer benötigt. Fast alle diese einfachen Begrenzer haben Sie bereits in Ihrem ersten Listing verwendet. Tabelle 2.3 fasst die wichtigsten Begrenzer zusammen:

Begrenzer	Bedeutung
Semikolon (;)	Dient als Abschluss einer Anweisung. Jeder Ausdruck, der mit einem Semikolon endet, wird als Anweisung behandelt. Der Compiler weiß dann, dass hier das Ende der Anweisung ist, und fährt nach der Abarbeitung dieser Anweisung mit der nächsten fort.
Komma (,)	Mit dem Komma trennen Sie gewöhnlich gleichartige Elemente, z. B. können Sie mehrere Variablen für Ganzzahlen so definieren: `int minSp, maxSp, startSp;` Andere Beispiele werden Sie jeweils im Kontext kennen lernen.
Geschweifte Klammern ({})	Zwischen den geschweiften Klammern wird ein Anweisungsblock zusammengefasst. In diesem befinden sich alle Anweisungen (abgeschlossen mit einem Semikolon), die ausgeführt werden sollen.
Gleichheitszeichen (=)	Das Gleichheitszeichen = steht in C für eine Zuweisung, z. B. in `int maxSpieler = 500;`

Tabelle 2.3 Einfache Begrenzer in C

2.5 Kommentare

Kommentare sind in einem C-Quelltext Textteile, die vom Compiler ignoriert werden. Sie können an einer beliebigen Stelle im Quelltext stehen und auf eine Programmzeile beschränkt sein oder sich über mehrere Zeilen erstrecken. Ihre Verwendung beeinflusst weder die Laufzeit des übersetzten Programms noch dessen Größe, weil die Kommentare bei der Übersetzung in Maschinencode entfernt werden.

Kommentare gibt es in zwei Ausführungen. Entweder schreiben Sie den Kommentar hinter zwei `//` oder zwischen `/*` und `*/`, zum Beispiel: `/* Bin ein Kommentar */`. In Kommentaren müssen Sie sich nicht an die Regeln der Zeichensätze halten und können beliebige Zeichen verwenden.

Sicherlich stellen Sie sich die Frage, welche Schreibweise Sie verwenden sollen. Hier kann ich zwar keine allgemeine Empfehlung abgeben, aber die Version mit `//` verwende ich persönlich am liebsten, um hinter einer Anweisung eine Zeile zu kommentieren. Die Form `/* Huhu */` nutze ich für einen mehrzeiligen Kommentar, die beispielsweise eine Funktion einleiten und als Ganzes erklären soll. Sie wird allerdings auch häufig verwendet, um einen Teil des Quellcode auszukommentieren. Hierzu ein paar Ausschnitte, die zeigen, wie Kommentare sinnvoll verwendet werden könnten:

```
/************************************/
/*       Ich beschreibe etwas       */
/*       Wert 1  =  ...             */
/*       Wert 2  =  ...             */
/*       Rückgabewert = ...         */
/************************************/

int fun = 1;  // Spaß muss immer auf 1 stehen

/*
printf( Der Code enthält einen Fehler );
printf Daher ist er auskommentiert;
*/
```

2.6 Kontrollfragen und Aufgaben

1. Welche der folgenden sind gültige Bezeichner in C?

```
anzahlPreise<30
_#Preise_kleiner_30
_groesster_Wert
groesster_Wert
größter_Wert
```

2. Korrigieren Sie das folgende Listing und bringen Sie es zur Ausführung:

```
00  // kap002/aufgabe001.c
01  #include <stdio.h>

02  int Main(void) {
03     printf('Ei-Pod\n');
04     return 0
05  }
```

3. Bei diesem Listing wurde etwas vergessen. Was fehlt hier?

```
00  // kap002/aufgabe002.c

01  int main(void) {
02     printf("Was ist hier falsch?\n");
03     printf("Kein Syntaxfehler!\n");
04     printf("Hier fehlt was...!\n");
05     return 0;
06  }
```

4. Erstellen Sie mit Steuerzeichen ein Programm, das mit maximal zwei `printf`-Befehlen folgende Ausgabe erzeugt:

```
        J
        u
        s
        t for F
              u
              n
```

Kapitel 3

Basisdatentypen in C

Ebenso wie Kochen nicht ohne Kochtopf geht, funktioniert eine Programmiersprache nicht ohne grundlegende Datentypen, mit denen man Daten in einer Variablen zwischenspeichern kann. Während Sie allerdings in einen Kochtopf alles Mögliche hineinschneiden können, müssen Sie bei den Basisdatentypen genau auf die »Zutaten« achten. In C sind hierfür Datentypen für Ganzzahlen, Fließkommazahlen und Zeichen vorhanden.

3.1 Variablen

Eine Variable ist im Grunde genommen nichts anderes als eine Adresse im Hauptspeicher. Dort legen Sie die Daten ab und greifen später, wenn Sie den Inhalt wieder benötigen, darauf zurück.

Um programmtechnisch ohne kryptische Adressangaben auf diese Adressen im Arbeitsspeicher zurückgreifen zu können, benötigen Sie einen eindeutigen Namen (Bezeichner) dafür. Der Compiler wandelt diesen später in eine Adresse um. Natürlich belegt jede dieser Variablen einen gewissen Speicherplatz. Wie viel das ist, hängt davon ab, welchen Datentyp Sie verwendet haben, wie viel Platz dieser auf einem bestimmten System beansprucht und mit welchen Werten er implementiert wurde. Der Standard schreibt hier nur eine Mindestgröße für die einzelnen Typen vor.

3.2 Deklaration und Definition

Bevor Sie eine Variable verwenden können, müssen Sie dem Compiler den Datentyp und den Bezeichner mitteilen. Dieser Vorgang wird als *Deklaration* bezeichnet. Was ein gültiger Bezeichner ist, haben Sie bereits in Abschnitt 2.4.1 erfahren. Die Datentypen lernen Sie in diesem Kapitel kennen.

Damit eine Variable auch verwendet werden kann, muss Speicherplatz dafür reserviert werden. Für das konkrete Speicherobjekt der Variablen im Programm bzw. im ausführbaren Code wird die *Definition* vereinbart.

Eine Deklaration darf mehrmals im Code vorkommen, eine Definition hingegen nur einmal im Programm. Wenn Sie beispielsweise eine Ganzzahlvariable `ivar` wie folgt vereinbaren:

```
int ivar;
```

deklarieren Sie eine Variable vom Datentyp `int` mit dem Bezeichner `ivar` und definieren sie auch gleichzeitig. Somit ist es in diesem Beispiel nicht falsch zu sagen, dass eine Definition gleichzeitig auch eine Deklaration ist.

Mehrere Bezeichner desselben Datentyps lassen sich auch in einer Zeile, getrennt durch Kommata, vereinbaren:

```
int ivar1, ivar2, ivar3;
```

Wenn Sie eine Variable nur deklarieren wollen, müssen Sie das Schlüsselwort `extern` davorsetzen:

```
// datei-01.c
extern int ivar;  // Deklaration
```

Mit diesem Schlüsselwort bestimmen Sie, dass kein Speicherplatz für `ivar` reserviert wird und dass Sie die Definition dieser Variablen (gewöhnlich) in einem anderen Modul vornehmen. Die Definition und Speicherplatzreservierung erfolgen jetzt beispielsweise in einem anderen Quelltextmodul:

```
// datei-02.c
int ivar;
```

Wozu überhaupt zwischen Deklaration und Definition unterscheiden?

An dieser Stelle mag Ihnen der Unterschied zwischen einer Deklaration und einer Definition noch etwas trivial erscheinen. Aber wenn Sie ihren Quellcode auf mehrere Quelltextmodule aufteilen, ist es essenziell, diesen Unterschied zu kennen und Deklarationen von Definitionen voneinander trennen zu können.

3.3 Initialisierung und Zuweisung von Werten

Nachdem Sie eine Variable definiert haben, besitzt diese noch keinen festen Wert (Ausnahmen: globale Variablen sowie lokale, mit `static` ausgezeichnete Variablen), sondern lediglich einen zufälligen Inhalt, der sich bereits in diesem Speicherbereich befunden hat (genauer gesagt, einen undefinierten Wert).

Einen Wert müssen Sie der Variablen erst noch zuweisen. Dies können Sie beispielsweise mit dem Zuweisungsoperator (=) oder mit einer Eingabefunktion wie etwa `scanf` erledigen:

```
int ivar;          // Definition
ivar = 12345;      // Zuweisung
```

Sie können aber auch gleich bei der Definition der Variablen einen Initial- bzw. Anfangswert zuweisen. Dieser Vorgang wird als *Initialisierung* bezeichnet. Im folgenden Beispiel wird einer Variablen mit dem Bezeichner `ivar` vom Datentyp `int` direkt bei der Definition der Wert 12345 zugewiesen:

```
int ivar = 12345;   // Initialisierung
```

Eine Initialisierung findet somit ausschließlich bei der Definition einer Variablen statt, wohingegen eine *Zuweisung* jederzeit und auch mehrmals notiert werden kann.

Variablen sofort mit einem Wert initialisieren

Damit Sie nicht versehentlich mit einer nicht initialisierten Variablen arbeiten, was zu undefinierten Ergebnissen führen könnte, sollten Sie es sich zur Gewohnheit machen, Variablen schon bei der Definition mit einem Anfangswert zu initialisieren, zum Beispiel:

```
int ivar = 0;  // ivar mit 0 initialisiert
```

3.4 Datentypen für Ganzzahlen

Für die Speicherung von vorzeichenbehafteten Ganzzahlen (hier zunächst: *standard signed integer types*) bietet C folgende in Tabelle 3.1 aufgelistete

Datentypen. Zusätzlich finden Sie in der Tabelle die **Mindestgrößen von Werten** und das **Formatzeichen**, um den Typ mit den Funktionen der `printf`-Familien formatiert auszugeben oder mit Funktionen der `scanf`-Familie einzulesen. Die tatsächlichen Wertebereiche sind besonders beim Typ `int` meistens größer.

Datentyp	Wertebereich (min.)	Formatzeichen
`signed char`	–128 +127	`%hhd` (für Dezimal) `%c` (für Zeichen)
`short`	–32768 +32767	`%hd` oder `%hi`
`int`	–32768 +32767	`%d` oder `%i`
`long`	–2.147.483.648 +2.147.483.647	`%ld` oder `%li`
`long long` (seit C99)	–9.223.372.036.854.775.808 +9.223.372.036.854.775.807	`%lld` oder `%lli`

Tabelle 3.1 Grundlegende Datentypen für Ganzzahlen

Neben diesen fünf `signed`-Ganzzahltypen (*standard signed integer types*) kann es abhängig von der Implementation auch erweiterte `signed`-Ganzzahltypen (*extended signed integer types*) wie etwa `__int128` geben, worauf hier allerdings nicht näher eingegangen wird.

Der bevorzugte Datentyp für Ganzzahlen lautet gewöhnlich `int`, weil per Definition die meisten Systeme damit am natürlichsten umgehen und häufig auch am schnellsten rechnen können. Benötigen Sie mehr, steht Ihnen `long` oder `long long` zur Verfügung. Bei kleineren Werten können Sie hingegen `short` verwenden. Eigentlich heißen die korrekten Typnamen `short int`, `int`, `long int` und `long long int`. In der Praxis wird normalerweise nur `short`, `long` und `long long` verwendet.

Überlauf der signed-Ganzzahlwerte

Bei `signed`-Ganzzahltypen kann es zu einem Werteüberlauf kommen. Der Standard schreibt nicht vor, wie das System auf einen Überlauf von `signed`-Ganzzahltypen reagieren soll. Deshalb lässt sich nicht voraussagen, wie sich das Programm weiter verhält (*undefined behavior*). Wenn Sie also zum maximalen Wert einer positiven Ganzzahl einen positiven Wert hinzuaddieren, befinden Sie sich plötzlich im negativen Bereich und haben einen Überlauf (engl. *Overflow*) erzeugt. Ein Beispiel:

```
int iVal = INT_MAX;  // INT_MAX enthält max. Wert für int
iVal += 2;  // Überlauf (undefined behavior)
```

Sie als Programmierer sind selbst dafür verantwortlich, dass es nicht so weit kommt.

Die Größe der Typen `int`, `short` und `long` ist nicht festgelegt. `int` ist mindestens so groß wie `short`. Der Datentyp `long` hingegen hat mindestens die Ausdehnung eines `int`. Daraus können Sie auch folgern, dass nicht hundertprozentig gesagt werden kann, wie breit ein bestimmter Typ implementiert ist. Allerdings können Sie sich mit Sicherheit auf folgende Reihenfolge verlassen:

```
signed char <= short <= int <= long <= long long
```

`<=` bedeutet hier: ist kleiner oder gleich.

Ähnlich sieht dies beim Abspeichern von einzelnen Bits aus. Auch hier hängt die Anzahl von Ihrem System ab. Der Standard schreibt hier ebenfalls nur vor, dass `signed char` der kleinste und `long long` der größte Typ für Ganzzahlen ist. Die anderen eingebauten Typen `short`, `int` und `long` liegen irgendwo dazwischen. Auf vielen Maschinen hat heute ein `char` 8 Bit und ein `long long` 64 Bit.

Das folgende Beispiel zeigt die einfache Ausgabe der vorzeichenbehafteten Ganzzahltypen mit `printf`:

```
00  // kap003/listing001.c
01  #include <stdio.h>

02  int main(void) {
03    signed char cVal = 100;
04    short sVal = 10000;
05    int iVal = 123456;
06    long lVal = 123456;
07    long long llVal = 123456;

08    printf("%hhd\n", cVal);
09    printf("%hd\n", sVal);
10    printf("%d\n", iVal);
11    printf("%ld\n", lVal);
12    printf("%lld\n", llVal);
13    return 0;
14  }
```

3.4.1 Vorzeichenlos und vorzeichenbehaftet

Für jeden der eben vorgestellten `signed`-Ganzzahltypen steht ein entsprechender `unsigned`-Ganzzahltyp (*standard unsigned integer type*) zur Verfügung. Wenn Sie beispielsweise eine Integer-Variable vereinbaren, ist diese (wenn auch implementationsabhängig) meistens vorzeichenbehaftet. Nehmen wir also an, Sie vereinbaren die folgende Variable:

```
int var;
```

Hier beträgt der Wertebereich von `int` abhängig von der Implementierung (siehe `INT_MIN` und `INT_MAX`) beispielsweise –2147483648 bis +2147483647. Mit dem Schlüsselwort `unsigned` können Sie jetzt eine ganzzahlige Variable ohne Vorzeichen vereinbaren. Das sieht beispielsweise so aus:

```
unsigned int var;
```

In diesem Fall könnten Sie keine negativen Werte mehr speichern. Dafür wird der positive Wertebereich von `int` (abhängig von der Implementierung von `UINT_MAX` in der Headerdatei `<limits.h>`) natürlich größer.

Größer ist nicht gleich größer

Damit Sie das jetzt nicht falsch verstehen: Der Datentyp `int` bleibt natürlich weiterhin breit. Mit `unsigned` verschiebt sich lediglich der Wertebereich (abhängig von der Implementierung) von beispielsweise –2147483647 bis +2147483647 auf mindestens 0 bis 4294967295.

Gleiches wie für `int` gilt auch für die Datentypen `short`, `long` und `long long`. Bei dem Datentyp `char` ist es etwas komplizierter. `char` kann entweder `signed char` oder `unsigned char` sein. Der Datentyp `char` wird gesondert in Abschnitt 3.5.1 behandelt.

Mit `signed` gibt es auch ein Schlüsselwort, um eine Variable explizit als vorzeichenbehaftet zu vereinbaren. Allerdings entspricht die Schreibweise von

```
signed int var;
```

der von

```
int var;
```

Somit ist die Verwendung des Schlüsselwortes `signed` überflüssig (außer bei `char`), weil ganzzahlige Datentypen ohne Verwendung von `unsigned` immer vorzeichenbehaftet sind.

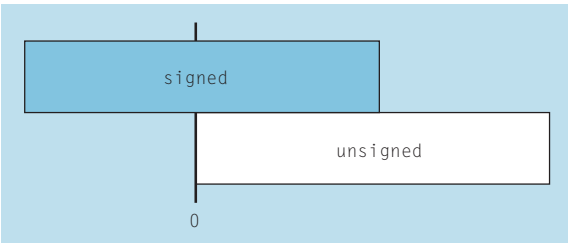

Abbildung 3.1 Mit »unsigned« ändert sich nicht die physikalische Größe, sondern es wird lediglich der Wertebereich verschoben.

Der Vollständigkeit halber finden Sie in Tabelle 3.2 einen Überblick über die `unsigned`-Gegenstücke bei den Ganzzahlen. Zusätzlich finden Sie den

Datentypen `_Bool` (seit C99), der ebenfalls zu den `unsigned`-Ganzzahltypen gehört. Die Wertebereiche in der Tabelle entsprechen auch hier wieder dem Mindestwert. Der tatsächliche Wert hängt von der Implementierung ab.

Datentyp	Wertebereich (min.)	Formatzeichen
`_Bool` (seit C99)	0 und 1	`%u`
`unsigned char`	0 bis 255	`%hhu` (für Dezimal) `%c` (für Zeichen)
`unsigned short`	0 bis 65.535	`%hu`
`unsigned int`	0 bis 65.535	`%u`
`unsigned long`	0 bis 4.294.967.295	`%lu`
`unsigned long long` (seit C99)	0 bis 18.446.744.073.709.551.615	`%llu`

Tabelle 3.2 Grunddatentypen für vorzeichenlose Ganzzahlen

Überlauf bei »unsigned«-Ganzzahlwerten

Bei `unsigned`-Ganzzahlwerten kann es nicht zu einem undefinierten Verhalten bei einem Überlauf kommen. Wenn hier der Wertbereich überschritten wird, wird mit einer sogenannten Modulo-Arithmetik weitergerechnet. Addieren Sie also zu einem maximalen `unsigned`-Ganzzahltypen 2 hinzu, wird mit 1 weitergerechnet. Ein Beispiel:

```
unsigned int uVal = UINT_MAX; // UINT_MAX enthält max. Wert
uVal += 2;  // = 1 (kein Überlauf)
```

Am Rande sei noch erwähnt, dass es auch hier – abhängig von der Implementation – erweiterte `unsigned`-Ganzzahltypen (*extended unsigned integer types*) geben kann (zum Beispiel `__uint128`).

Ganzzahltypen mit fester Breite

Seit C99 werden diese Datentypen über die plattformunabhängige Headerdatei `<stdint.h>` um Ganzzahltypen wie etwa `int8_t`, `int16_t`, `int32_t`, `int64_t` usw. mit fester Länge ergänzt.

3.4.2 Suffixe für Ganzzahlen

Wenn Sie den Compiler informieren müssen, wie er ein bestimmtes Literal interpretieren soll oder von welchem Typ dieses sein soll, können Sie ein dem Typ entsprechendes Präfix oder Suffix zum Literal hinzufügen. Das Suffix `u` oder `U` deutet beispielsweise an, dass es sich um eine vorzeichenlose (`unsigned`) Zahl handelt. `l` bzw. `L` gibt an, dass es sich um eine `long`-Zahl handelt. Den Datentyp `long long` zeigen Sie mit `ll` bzw. `LL` an. Das können Sie auch mit `unsigned` kombinieren, indem Sie ein `ul` oder `UL` für `unsigned long` und `ull` oder `ULL` für `unsigned long long` verwenden. Verzichten Sie auf das Suffix, verwendet der Compiler `int`, wenn der Wert passt.

Ein Beispiel:

```
unsigned int uVal = 1000u;
long lVal = 100000L;
unsigned long ulVal = 2222222UL;
long long llVal = 1234LL;
unsigned long long ullVal = 12341234323ULL;
unsigned int uHexVal = 0X42U;
```

3.5 Datentyp für Zeichen

In diesem Abschnitt erfahren Sie etwas zu den Zeichentypen in C. Zwar werden Sie in diesem Buch vorwiegend mit dem Datentyp `char` arbeiten, trotzdem sollen Sie auch kurz erfahren, welche Möglichkeiten es gibt, erweiterte bzw. umfangreichere Zeichensätze zu verwenden. Beachten Sie, dass die Verwendung von erweiterten Zeichensätzen (wie Unicode) jenseits von `char` kein triviales Thema mehr ist. Für Sie ist es daher zunächst einmal wichtig, dass Sie sich mit dem Datentypen `char` vertraut machen.

Wenn Sie anfangen, internationale Programme zu schreiben, dann werden Sie sich auch intensiver mit dem Thema »Unicode« befassen müssen.

3.5.1 Der Datentyp char

Der grundlegende Datentyp für Zeichen lautet `char` und belegt gewöhnlich ein Byte an Speicherplatz, was somit meistens (nicht immer) $2^8 = 256$ Ausprägungen ermöglicht. Der Datentyp `char` ist zumindest groß genug, dass alle Zeichen des Basis-Ausführungszeichensatzes darin gespeichert werden können. Wird außerdem ein Zeichen in einem `char` gespeichert, dann ist es garantiert, dass der gespeicherte Wert der nichtnegativen Kodierung im Zeichensatz entspricht. Ein Zeichensatz wiederum ist dazu da, einem Zeichen einen bestimmten Wert zuzuweisen, weil ein Rechner letztendlich nur Dualzahlen speichern kann. Ein Beispiel:

```
char ch = 'A';  // Dezimal 65 im ASCII-Zeichensatz
```

Im weit verbreiteten ASCII-Zeichensatz entspricht die Zeichenkonstante `'A'` dem Dezimalwert 65. So wäre es beispielsweise auch möglich, stattdessen Folgendes anzugeben:

```
char ch = 65;  // Entspricht dem Zeichen 'A' (ASCII)
```

Das ist ohne Probleme möglich, weil `char` ja auch ein Integertyp ist. In der Praxis ist die Verwendung eines Dezimalwerts anstelle einer Zeichenkonstante allerdings nicht zu empfehlen, wenn Sie `char` als Zeichentyp und nicht als Integertyp verwenden wollen. Zum einen lässt sich nicht sofort erkennen, welches Zeichen hier dahintersteckt, und zum anderen schreibt der Standard nicht vor, welcher Zeichensatz verwendet werden soll. Zwar dürfte zu 99,9 % die ASCII-Zeichensatztabelle zum Einsatz kommen, aber wenn dann doch einmal ein anderer Zeichensatz verwendet wird, sind Sie mit der Zeichenkonstante immer auf der sicheren Seite. Hierzu ein einfaches Beispiel:

```
00  // kap003/listing002.c
01  #include <stdio.h>

02  int main(void) {
```

```
03    char ch01 = 'A'; // bei ASCII 65
04    char ch02 = 66;  // besser wäre 'B'
05    printf("Dezimal: %d; Zeichen: %c\n", ch01, ch01);
06    printf("Dezimal: %d; Zeichen: %c\n", ch02, ch02);
07    return 0;
08  }
```

Das Programm bei der Ausführung:

```
Dezimal: 65; Zeichen: A
Dezimal: 66; Zeichen: B
```

`char` ist auch ein Ganzzahldatentyp, mit dem Sie rechnen können. Aufgrund des kleineren Wertebereichs wird dieser Typ dazu jedoch relativ selten genutzt.

Vorzeichen bei »char«

Es hängt von der Compilerimplementierung ab, ob `char` auch negative Zahlen aufnehmen kann – ob `char` also als `signed char` oder `unsigned char` implementiert ist. Dies ist unter anderem dann wichtig zu wissen, wenn Sie `char` als Typ für Ganzzahlen verwenden wollen.

3.5.2 Der Datentyp wchar_t

Für die Zeichensätze mancher Sprachen wie etwa der chinesischen mit über tausend Zeichen ist der Datentyp `char` zu klein. Für die Darstellung beliebiger landesspezifischer Zeichensätze kann daher der Breitzeichentyp `wchar_t` (*wide char* = breite Zeichen) aus der Headerdatei `<stddef.h>` verwendet werden. Bei der Verwendung eines solchen Zeichens muss vor die einzelnen Hochkommata noch das Präfix `L` gestellt werden:

```
wchar_t ch = L'Z';
```

Entsprechend wird auch bei dem Formatzeichen für die Ausgabe oder Eingabe eines `wchar_t` ein `l` vor das `c` gesetzt (`%lc`):

```
wprintf("%lc", ch);
```

Auch die üblichen Funktionen, die mit Zeichenketten arbeiten, können Sie mit `wchar_t` nicht mehr verwenden, und Sie müssen stattdessen auf die entsprechenden `w`-Versionen (wie beispielsweise `wprintf()`) zurückgreifen.

Die Größe von `wchar_t` lässt sich nicht exakt beschreiben, meistens beträgt sie jedoch 2 oder 4 Bytes. Es lässt sich lediglich mit Sicherheit sagen, dass `wchar_t` mindestens so groß wie `char` und höchstens so groß wie `long` ist. `wchar_t` muss auf jeden Fall mindestens groß genug sein, um alle Werte des größten unterstützten Zeichensatzes aufnehmen zu können.

Zeichen und Zeichensatz

Egal, welchen Zeichentyp Sie verwenden, Sie sollten sich immer vor Augen halten, dass `char` und `wchar_t` selbst keine Zeichen speichern, sondern letztendlich nur Ganzzahlen, die ihre Bedeutung erst mit dem auf dem Rechner befindlichen Zeichensatz erhalten.

3.5.3 Unicode-Unterstützung

Der Unicode-Standard definiert mit UTF-8, UTF-16 und UTF-32 drei Zeichenkodierungsformate. Jedes Format hat seine Vor- und Nachteile. Bisher haben Programmierer `char` verwendet, um UTF-8 zu nutzen, `unsigned short` oder `wchar_t` für UTF-16 und `unsigned long` oder `wchar_t` für UTF-32. Mit dem neuen C11-Standard können Sie jetzt zwei Typen mit einer plattformunabhängigen Breite mit `char16_t` und `char32_t` für UTF-16 und UTF-32 aus der Headerdatei `<uchar.h>` nutzen.

```
#include <uchar.h>

char utf8ch = u8'Z';        // UTF-8
char16_t utf16ch = u'Z';    // UTF-16
char32_t utf32ch = U'Z';    // UTF-32
```

Für UTF-8 können Sie nach wie vor `char` verwenden. C11 hat außerdem die Präfixe `u` und `U` für UTF-16- bzw. UTF-32-Literale und das Präfix `u8` für UTF-8-Literale eingeführt. Auch Unicode-Konvertierungsfunktionen sind in `<uchar.h>` deklariert.

3.6 Datentypen für Fließkommazahlen

Fließkommaliterale sind Werte mit einer Nachkommastelle und enthalten ein Dezimaltrennzeichen in Form eines Punktes (beispielsweise 3.1415, .25, 33. usw.). Auch eine wissenschaftliche Schreibweise des Exponenten als Zehnerpotenz ist möglich (22.44e–3 beispielsweise entspricht 0.02244). Im Standard finden Sie die folgenden Fließkommatypen:

Datentyp	Wertebereich	Formatzeichen	Genauigkeit
`float`	1.2E–38 3.4E+38	`%f`	einfach
`double`	2.3E–308 1.7E+308	`%f` (`%lf` für scanf)	doppelt
`long double`	3.4E–4932 1.1E+4932	`%Lf`	zusätzlich

Tabelle 3.3 Datentypen für Fließkommazahlen

Beachten Sie, dass die Genauigkeit und die Wertebereiche dieser Typen komplett implementierungsabhängig sind. Es ist lediglich gewährleistet, dass bei `float`, `double` und `long double` (hier von links nach rechts) jeder Typ den Wert des vorherigen aufnehmen kann. Auf den meisten Architekturen entsprechen `float` und `double` den IEEE-Datentypen. Die Norm IEEE 754 definiert dabei die Darstellungen für binäre Gleitkommazahlen im Computer.

Bevorzugter Fließkommatyp

In der Praxis empfiehlt es sich, immer den Fließkommatyp `double` zu verwenden, weil der Compiler den Typ `float` intern häufig ohnehin in den Typ `double` umwandelt.

Im Gegensatz zu den Ganzzahlen gibt es bei den Fließkommazahlen keine Unterschiede zwischen vorzeichenbehafteten und vorzeichenlosen Zahlen. In C++ sind alle Fließkommazahlen vorzeichenbehaftet.

Beachten Sie außerdem, dass die Trennzeichen bei den Fließkommazahlen in US-amerikanischer Schreibweise angegeben werden. Anstatt eines Kommas zwischen den Zahlen müssen Sie einen Punkt setzen (man spricht im Englischen von *floating* **point** *variables*):

```
double pi = 3,14159265;   // FALSCH
double pi = 3.14159265;   // RICHTIG
```

Wenn einer der Werte vor oder nach dem Dezimalpunkt 0 ist, beispielsweise 0.5 oder 1.0, können Sie die Ziffer 0 auch weglassen. Der Compiler ergänzt die 0 automatisch.

```
double c = .5;  // entspricht 0.5
double d = 5.;  // entspricht 5.0
```

»float_t« und »double_t«

Erwähnt werden sollten an dieser Stelle noch die Gleitkommatypen `float_t` und `double_t` aus der Headerdatei `<math.h>`, die seit C99 vorhanden sind, mit denen bei arithmetischen Operationen intern gerechnet wird und durch welche die notwendigen Konvertierungen entfallen. Welcher Typ dabei verwendet wird, hängt vom Wert des Makros `FLT_EVAL_METHOD` ab.

3.6.1 Suffixe für Fließkommazahlen

Wie den Ganzzahlen können Sie den Fließkommazahlen ebenfalls ein Suffix hinzufügen. Mit dem Suffix `f` oder `F` kennzeichnen Sie eine Fließkommazahl mit einer einfachen Genauigkeit (`float`). Das Suffix `l` oder `L` hingegen deutet auf eine Fließkommazahl mit erhöhter Genauigkeit hin (`long double`). Ohne eine Angabe bei Fließkommazahlen wird der Typ `double` verwendet. Die Verwendung des Exponenten, mit dem die Größe der Zahl als Zehnerpotenz angegeben wird, kann mit `e` oder `E` (beide Buchstaben haben dieselbe Bedeutung), gefolgt von einem optionalen `+` oder `-` und einer Ziffernsequenz, notiert werden (22.45e0 etwa entspricht 22.45, 22.45e1 entspricht 224.5, und 22.45e–3 ist 0.02245). Hier einige Beispiele:

```
float fVal = 3.33f;
long double ldVal = 32.32L;
double eVal1 = 4.3e10;
double eVal2 = 3.4e-5;
long double eVal3 = 8.4e123L;
```

3.6.2 Komplexe Gleitkommatypen

Mit dem C99-Standard wurden auch komplexe Gleitkommatypen in der Headerdatei `<complex.h>` implementiert. Eine komplexe Zahl wird hierbei als Paar aus Real- und Imaginärteil dargestellt, die auch mit den Funktionen `creal()` und `cimag()` ausgegeben werden können. Beide Teile der komplexen Zahl bestehen entweder aus den Typen `float`, `double` oder `long double`. Daher gibt es wie bei den reellen Gleitpunktzahlen die folgenden drei komplexen Gleitkommatypen:

```
float _Complex
double _Complex
long double _Complex
```

Um die umständliche Schreibweise mit dem Unterstrich `_Complex` zu vermeiden, ist in der Headerdatei `<complex.h>` das Makro `complex` definiert. Anstelle des Schlüsselwortes `_Complex` können Sie auch `complex` verwenden:

```
float complex       // gleich wie float _Complex
double complex      // gleich wie double _Complex
long double complex // gleich wie long double _Complex
```

Da komplexe Zahlen einen Real- und einen Imaginärteil haben, beträgt die Größe des Datentyps in der Regel das Doppelte der Größe der grundlegenden Datentypen. Ein `float _Complex` benötigt somit 8 Bytes, weil im Grunde genommen zwei `float`-Elemente benötigt werden. Folgendes Listing soll das verdeutlichen:

```
00 // kap003/listing003.c
01 #include <stdio.h>
02 #include <complex.h>
```

```
03  int main(void) {
04    float complex fc = 2.0 + 3.0*I;
05    // 4 Bytes
06    printf("sizeof(float) : %zu\n", sizeof(float));
07    // 8 Bytes (realer und imaginärer Teil)
08    printf("sizeof(float complex) : %zu\n",
                    sizeof(float complex));
09    // Ausgabe von Real- und Imaginärteil
10    printf("%f + %f\n", creal(fc), cimag(fc));
11    return 0;
12  }
```

Des Weiteren ist in der Headerdatei das Makro `I` definiert, das die imaginäre Einheit mit dem Typ `const float complex` darstellt. Vielleicht hier eine kurze Hintergrundinformation zu komplexen Gleitpunktzahlen – eine komplexe Zahl `zVal` wird beispielsweise folgendermaßen in einem kartesischen Koordinatensystem dargestellt:

```
zVal = xVal + yVal * I
```

`xVal` und `yVal` sind dabei reelle Zahlen, und `I` ist der imaginäre Teil. Die Zahl `xVal` wird hier als realer Teil betrachtet, und `yVal` ist der imaginäre Teil von `zVal`.

Ebenfalls in C99 eingeführt wurden Gleitkomma-Datentypen für rein imaginäre Zahlen mit `float _Imaginary`, `double _Imaginary` und `long double _Imaginary`.

3.7 Boolescher Datentyp

Im C99-Standard wurde mit `_Bool` ein boolescher Wahrheitswert eingeführt, der auf jeden Fall groß genug ist, um die Werte 0 und 1 aufzunehmen. Der Datentyp `_Bool` gehört zur Gruppe der `unsigned`-Ganzzahltypen.

Glücklicherweise existiert für den Typ `_Bool` (ebenfalls seit C99) in der Headerdatei `<stdbool.h>` das Makro `bool`, sodass Sie den Bezeichner `bool` wie in C++ verwenden können. Allerdings müssen Sie hierfür die Headerdatei `<stdbool.h>` inkludieren.

Boolesche Werte sind Elemente einer »booleschen Algebra«, die einen von zwei möglichen Werten annehmen können. Dieses Wertepaar hängt von der Anwendung ab und lautet entweder wahr/falsch, `true`/`false` oder eben ungleich 0/gleich 0. In C kann hierfür das Wertepaar `true` (für wahr) und `false` (für falsch) verwendet werden, das in der Headerdatei `<stdbool.h>` definiert ist.

Sie können auch das Paar ungleich 0 (für wahr) und gleich 0 (für falsch) als Dezimalwert verwenden:

```
#include <stdbool.h>
// ...
// Schalter auf Wahr setzen
_Bool b1 = 1;
// Schalter auf Unwahr setzen
_Bool b2 = 0;
// Benötigt <stdbool.h>
bool b3 = true; // wahr
// Benötigt <stdbool.h>
bool b4 = false; // unwahr
```

Um hier kein Durcheinander zu verursachen, muss noch erwähnt werden, dass der C99-Standard den Typ `_Bool` als echten Datentyp implementiert hat. Das Makro `bool` und das Wertepaar `true` bzw. `false` können Sie nur verwenden, wenn Sie die Headerdatei `<stdbool.h>` inkludieren.

3.8 Speicherbedarf mit sizeof ermitteln

Wenn Sie die Größe eines Typs benötigen, wird der `sizeof`-Operator verwendet. Dieser gibt in der Regel die Größe des Operanden in Byte(s) zurück und wird beispielsweise bei der dynamischen Speicherreservierung verwendet oder wenn Sie Programme schreiben, die auf andere Plattformen portierbar sind. Als Rückgabetyp von `sizeof` ist `size_t` definiert. `size_t` ist ein implementierungsabhängiger `unsigned`-Ganzzahlentyp und in der Headerdatei `<stddef.h>` (und anderen Headerdateien) definiert. Wie viel Speicherplatz ein Variablentyp letztendlich benötigt, ist

wie immer implementierungsabhängig. Sicher ist nur, dass `sizeof(char)` stets den Wert 1 zurückgibt. Die Formatanweisung für `size_t` lautet `%zu`.

Hierzu ein einfaches und im Augenblick recht sinnfreies Beispiel, in dem der `sizeof`-Operator verwendet wird:

```
00 // kap003/listing004.c
01 #include <stdio.h>

02 int main(void) {
03   int ival = 0;
04   double dval = 0;
05   printf("sizeof(ival)  : %zu\n", sizeof(ival));
06   printf("sizeof(dval)  : %zu\n", sizeof(dval));
07   // So geht es auch
08   printf("sizeof(float) : %zu\n", sizeof(float));
09   size_t sz = sizeof(char);
10   printf("sizeof(char)  : %zu\n", sz);
11   return 0;
12 }
```

Das Programm sieht bei der Ausführung auf meinem System folgendermaßen aus:

```
sizeof(ival)  : 4
sizeof(dval)  : 8
sizeof(float) : 4
sizeof(char)  : 1
```

Speicherausrichtung eines Operanden ermitteln

Seit C11 gibt es den Operator `_Alignof`, mit dem Sie die Speicherausrichtung des Operanden ermitteln können. Dieser Operand ist auch als Makro mit `alignof` vorhanden, damit Sie ihn etwa mit `alignof(long double)` so komfortabel wie schon den `sizeof`-Operator verwenden können. Wie der `sizeof`-Operator liefert auch `alignof` die Speicherausrichtung des Operanden vom Typ `size_t` zurück.

3.9 Wertebereiche der Datentypen ermitteln

Der Standard selbst legt den Wertbereich und die Größe der einzelnen Datentypen nicht fest, sondern schreibt nur die Relation zwischen den Größen der Datentypen vor. Für jeden Basisdatentyp werden lediglich Mindestgrößen gefordert. Dadurch ergeben sich für den Compiler verschiedene Gestaltungsmöglichkeiten.

Eine Möglichkeit dabei ist, dass der Datentyp `int` so festgelegt wird, dass seine Größe der Datenwortgröße des Prozessors entspricht. Das handhaben aber nicht alle Compiler so, und es gibt auch andere Schemas. Die Größe des Zeigertyps wiederum richtet sich häufig nach der Größe des Speicherbereichs, der vom Programm aus erreichbar (adressierbar) sein muss. Es ist daher durchaus möglich, dass der Speicherbereich kleiner oder größer als die Prozessorarchitektur ist.

Wenn man dann davon ausgeht, dass ein Byte 8 Bit groß ist, was auf vielen Architekturen der Fall ist, dann sind die anderen Datentypen alle ein Vielfaches von 8 Bit. Aufgrund dieser Möglichkeiten ergeben sich verschiedene sogenannte *Datenmodelle* (oder auch Programmiermodelle). Ich will an dieser Stelle nicht näher auf die verschiedenen Datenmodelle eingehen; es geht mir vielmehr darum, dass Sie verstehen, warum die Datentypen auf unterschiedlichen Systemen unterschiedliche Wertebereiche haben können.

In Tabelle 3.4 finden Sie eine Übersicht über die gängigen Datenmodelle.

Modell	char	short	int	long	long long	void*
IP16	8	16	16	32	64	16
LP32	8	16	16	32	64	32
ILP32	8	16	32	32	64	32
LLP64	8	16	32	32	64	64
LP64	8	16	32	64	64	64

Tabelle 3.4 Bits von Datentypen verschiedener Datenmodelle

Modell	char	short	int	long	long long	void*
ILP64	8	16	64	64	64	64
SILP64	8	64	64	64	64	64

Tabelle 3.4 Bits von Datentypen verschiedener Datenmodelle (Forts.)

Wenn Sie wissen wollen, welche implementierungsabhängigen Wertebereiche die einzelnen Datentypen denn auf dem auszuführenden System haben, dann finden Sie die vom Compilerhersteller vergebenen Größen in der Headerdatei `<limits.h>` für Integertypen und `<float.h>` für Gleitkommatypen. Benötigen Sie hingegen Integertypen mit einer festen Größe, dann bietet Ihnen der Standard entsprechende Typen wie `int8_t`, `int16_t` usw. an, die in der Headerdatei `<stdint.h>` definiert sind.

3.9.1 Limits von Integertypen

Möchten Sie erfahren, welchen Wertebereich `int` oder die anderen Ganzzahl-Datentypen auf Ihrem System haben, finden Sie in der Headerdatei `<limits.h>` entsprechende Konstanten dafür. Für den Datentyp `int` beispielsweise finden Sie die Konstanten `INT_MIN` für den minimalen und `INT_MAX` für den maximalen Wert.

Um diese Werte zu ermitteln, müssen Sie selbstverständlich auch den Header `<limits.h>` im Programm inkludieren. Das folgende Listing gibt Ihnen den tatsächlichen Wertebereich für die Datentypen `char`, `short`, `int`, `long` und `long long` auf Ihrem System aus:

```
00  // kap003/listing005.c
01  #include <stdio.h>
02  #include <limits.h>

03  int main(void) {
04    printf("min. char-Wert     : %d\n", SCHAR_MIN);
05    printf("max. char-Wert     : +%d\n", SCHAR_MAX);
06    printf("min. short-Wert    : %d\n", SHRT_MIN);
07    printf("max. short-Wert    : +%d\n", SHRT_MAX);
```

```
08    printf("min. int-Wert      : %d\n", INT_MIN);
09    printf("max. int-Wert      : +%d\n", INT_MAX);
10    printf("min. long-Wert     : %ld\n", LONG_MIN);
11    printf("max. long-Wert     : +%ld\n", LONG_MAX);
12    printf("min. long long-Wert: %lld\n", LLONG_MIN);
13    printf("max. long long-Wert: +%lld\n", LLONG_MAX);
14    return 0;
15  }
```

Die Ausgabe des Programms hängt natürlich von der Implementierung ab. Bei mir sieht sie wie folgt aus:

```
min. char-Wert     : -128
max. char-Wert     : +127
min. short-Wert    : -32768
max. short-Wert    : +32767
min. int-Wert      : -2147483648
max. int-Wert      : +2147483647
min. long-Wert     : -2147483648
max. long-Wert     : +2147483647
min. long long-Wert: -9223372036854775808
max. long long-Wert: +9223372036854775807
```

Einen Überblick über alle Konstanten in der Headerdatei `<limits.h>` und deren Bedeutungen finden Sie auf den entsprechenden Manpages, in der Onlinehilfe des Compilers oder im Web unter *http://en.cppreference.com/w/c/types/limits*. Und auch hier nicht zu vergessen das Committee Draft des C11-Standard unter *http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1570.pdf*. Dasselbe gilt auch für die gleich folgenden Limits von Gleitkommazahlen.

3.9.2 Limits von Fließkommazahlen

Auch bei Fließkommazahlen gibt es eine Headerdatei mit Konstanten, in der Sie die Wertebereiche ermitteln können. Alle implementierungsabhängigen Wertebereiche für Fließkommazahlen sind in der Headerdatei `<float.h>` deklariert. Zur Demonstration zeige ich Ihnen nachfolgend ein

einfaches Listing. Dieses ermittelt die Genauigkeit der Dezimalziffern aus den Konstanten FLT_DIG (für float), DBL_DIG (für double) und LDBL_DIG (für long double), die im Header <float.h> deklariert sind:

```
00  // kap003/listing006.c
01  #include <stdio.h>
02  #include <float.h>

03  int main(void) {
04    printf("float Genauigkeit      : %d\n", FLT_DIG);
05    printf("double Genauigkeit     : %d\n", DBL_DIG);
06    printf("long double Genauigkeit: %d\n", LDBL_DIG);
07    return 0;
08  }
```

So sieht die Ausgabe des Programms bei mir aus:

```
float Genauigkeit      : 6
double Genauigkeit     : 15
long double Genauigkeit: 15
```

3.9.3 Integertypen mit fester Größe verwenden

Wenn Sie sich nicht auf die implementierungsabhängigen Größen der Basis-Integertypen auf den verschiedenen Systemen verlassen wollen/können bzw. einen Integertypen mit einer *vorgegebenen Breite* benötigen, finden Sie seit C99 entsprechende Typen in der Headerdatei <stdint.h> definiert. Mit »vorgegebener Breite« ist die Anzahl der Bits zur Darstellung des Wertes gemeint. Die speziellen Formatierungsspezifizierer für die printf- und scanf-Familien hingegen finden Sie in der Headerdatei <inttypes.h>. Die einzelnen Typen können Sie hierbei in folgende Gruppen aufteilen (*N* steht für die Anzahl von Bits, und Typen mit u (unsigned) sind vorzeichenlos):

- intN_t und uintN_t: ein Integertyp mit einer Breite von exakt *N* Bits wie beispielsweise int64_t bzw. uint64_t für einen Integertypen mit 64 Bit Breite. Entsprechend den Typen finden Sie in der Headerdatei <stdint.h> auch die zugehörigen Limits für die minimalen und maximalen Werte mit INTN_MIN und INTN_MAX bzw. UINTN_MAX.

- `int_least`*N*`_t` und `uint_least`*N*`_t`: ein Integertyp mit einer Breite von mindestens *N* Bits. Er ist damit garantiert der kleinste Typ der Implementation. Auch dazu finden Sie mit `INT_LEAST`*N*`_MIN` und `INT_LEAST`*N*`_MAX` bzw. `UINT_LEAST`*N*`_MAX` entsprechende Limits für die minimalen bzw. maximalen Werte. *N* kann hierbei 8, 16, 32, 64 sein.
- `int_fast`*N*`_t` und `uint_fast`*N*`_t`: ein schneller Integertyp mit einer Breite von mindestens *N* Bits. Dieser Typ ist garantiert der schnellste Integertyp in der Implementation. Auch hierzu finden Sie mit `INT_FAST`*N*`_MIN` und `INT_FAST`*N*`_MAX` bzw. `UINT_FAST`*N*`_MAX` entsprechende Limits für die minimalen bzw. maximalen Werte. *N* kann hierbei 8, 16, 32, 64 sein.
- `intmax_t` und `uintmax_t`: der garantiert größtmögliche Integertyp der Implementation. Den minimalen und maximalen Wert können Sie mit `INTMAX_MIN` und `INTMAX_MAX` bzw. `UINTMAX_MAX` ermitteln.

Wenn Sie die Headerdatei `<stdint.h>` eingebunden haben, können Sie diese Integertypen mit fester Bitbreite genauso einsetzen und verwenden wie die Integertypen ohne feste Größe:

```
00  // kap003/listing007.c
01  #include <stdio.h>
02  #include <stdint.h>

03  int main(void) {
04    int64_t bigVar = 12345678;
05    printf("bigVar          : %lld\n", bigVar);
06    printf("sizeof(int64_t) : %zu\n", sizeof(int64_t));
07    printf("INT64_MAX       : %lld\n", INT64_MAX);
08    return 0;
09  }
```

Das Beispiel bei der Ausführung:

```
bigVar          : 12345678
sizeof(int64_t) : 8
INT64_MAX       : 9223372036854775807
```

3.9.4 Sicherheit beim Kompilieren mit _Static_assert

Mit dem neuen C11-Feature `_Static_assert()` überprüfen Sie einen konstanten Ausdruck zwischen den Klammern zur Übersetzungszeit. Trifft die Auswertung des Ausdrucks nicht zu, bricht der Compiler mit einer Fehlermeldung ab, die Sie ebenfalls mit angeben können. Auch hier wurde mit `static_assert()` ein komfortables Makro definiert, um nicht die Schreibweise mit dem vorangestellten Unterstrich verwenden zu müssen. Damit Sie dieses Feature verwenden können, müssen Sie die Headerdatei `<assert.h>` einbinden. Ein einfaches Beispiel hierzu:

```
// kap003/listing008.c
#include <assert.h>
…
static_assert( sizeof(long double) == 16,
               "Need 16 byte long double" );
```

Hier fordern wir den Compiler auf, den Ausdruck `sizeof(long double) == 16` zu überprüfen. Unsere Anwendung erfordert 16 Bytes für ein `long double` auf dem System, auf dem der Quellcode übersetzt wird. Ist der Ausdruck wahr, wird der Quellcode weiter übersetzt. Ist der Ausdruck hingegen falsch, bricht der Compiler die Übersetzung ab und gibt die dahinter angegebene Fehlermeldung aus (hier: `Need 16 byte long double`).

Wollen Sie beispielsweise sichergehen, dass Ihr Programm **nicht** auf einem System übersetzt wird, auf dem `unsigned char` **mehr** als 8 Bit hat, können Sie dies mit `static_assert()` folgendermaßen umsetzen:

```
// kap003/listing009.c
#include <assert.h>  // für static_assert
#include <limits.h>  // für CHAR_BIT
…
static_assert( CHAR_BIT == 8,
               "unsigned char hat hier nicht 8 Bit!" );
```

Die Verwendung von `static_assert()` kann nur empfohlen werden. Sie kostet überhaupt keine Laufzeit der Anwendung, weil diese Sicherheitschecks nur vom Compiler benutzt werden. Lediglich die Übersetzungszeit

nimmt logischerweise zu. Aber ich denke, damit kann man leben, weil hiermit auf so manche Sicherheitsüberprüfung während der Laufzeit des Programms – und dadurch auf Code – verzichtet werden kann, was wiederum die Laufzeit verbessert.

Mithilfe von `static_assert()` können Sie unter Umständen Fehler abfangen, die Sie mit Präprozessoranweisungen wie mit den Direktiven `#if` und `#error` unmöglich ermitteln können, weil `static_assert()` zur Compilerzeit ausgeführt wird und der Präprozessor eben vor der Compilerzeit.

3.10 Konstanten erstellen

Benötigen Sie einen unveränderbaren Wert, können Sie eine Konstante verwenden. Der Sinn und Zweck einer solchen Konstante ist es, dass der Wert zur Laufzeit des Programms nicht mehr verändert werden kann. Das Gegenstück zu einer Konstante ist eine Variable. Eine Konstante definieren Sie, indem Sie vor den eigentlichen Datentyp das Schlüsselwort `const` setzen:

```
const int cIvar = 365;      // Integerkonstante
const char dquote = '"';    // Zeichenkonstante
const float pi = 3.141592;  // Fließkommakonstante
```

Wenn Sie nun aus Versehen versuchen, den Wert der Konstante zu verändern, gibt der Compiler zur Übersetzungszeit den Fehler aus. Somit können `const`-Werte auf gewöhnlichem Wege der direkten Zuweisung nicht mehr verändert werden. Wenn Sie beim folgenden Beispiel versuchen, den Wert von `cIvar` zu ändern, wird sich der Compiler mit einer Fehlermeldung bei Ihnen beschweren:

```
const int cIvar = 365;  // Konstante
cIvar = 364;            // Fehler, da const
```

In der Praxis werden Konstanten mit `const` recht häufig bei Variablen, Zeigern, Parametern von Funktionen usw. verwendet – eben immer, wenn ein Wert nicht mehr verändert werden soll.

Qualifikator

Bei dem Schlüsselwort `const` handelt es sich um einen Qualifikator. Über solche Qualifikatoren kann ein Datentyp bei der Vereinbarung modifiziert werden. Neben `const` gibt es noch weitere Qualifikatoren wie `volatile`, `restrict` (seit C99) oder `_Atomic` (seit C11).

3.11 Lebensdauer und Sichtbarkeit von Variablen

Die **Lebensdauer** einer Variablen gilt immer bis zum Ende des Anweisungsblocks, in dem sie definiert wurde. Ein einfaches Beispiel:

```
00  // kap003/listing010.c
...
01  {  // Anweisungsblock - Anfang
02    int ivar = 1234;
03    printf("%d\n", ivar);
04  }  // Anweisungblock - Ende
05  printf("%d\n", ivar);  // Fehler!!!
...
```

Im Beispiel wurde innerhalb des Anweisungsblocks in der Zeile **(02)** eine Integervariable mit dem Bezeichner `ivar` vereinbart. Die Ausgabe in der Zeile **(03)** wäre noch in Ordnung, aber bei der Ausgabe in der Zeile **(05)** wird sich der Compiler mit einer Fehlermeldung beschweren, weil hier die Variable `ivar` nicht mehr gültig ist. Die Variable `ivar` ist nur innerhalb des Anweisungsblocks der Zeilen **(01)** bis **(04)** gültig. Danach existiert sie nicht mehr. Das Problem in diesem Beispiel könnten Sie beheben, indem Sie die Variable `ivar` außerhalb, also vor dem Anweisungsblock in der Zeile **(01)** notieren:

```
00  // kap003/listing011.c
...
01  int ivar = 1234;
02  {  // Anweisungsblock - Anfang
```

```
03    printf("%d\n", ivar);  // = 1234
04  }  // Anweisungblock - Ende
05  printf("%d\n", ivar);    // = 1234
…
```

Wie in der Zeile **(01)** vereinbart, ist die **Sichtbarkeit** der Variablen `ivar` jetzt sowohl in der Zeile **(03)** als auch in der Zeile **(05)** gegeben. Existiert nun allerdings im Anweisungsblock eine weitere Variable `ivar`, greift der innere Block auf die nächstliegende gleichnamige Variable im inneren Block zu, beispielsweise:

```
00  // kap003/listing012.c
…
01  int ivar = 1234;
02  {  // Anweisungsblock - Anfang
03    int ivar = 4321;
04    printf("%d\n", ivar);  // = 4321
05  }  // Anweisungblock - Ende
06  printf("%d\n", ivar);    // = 1234
…
```

Durch die Vereinbarung einer weiteren gleichnamigen Variablen `ivar` in der Zeile **(03)** wird innerhalb des Anweisungsblocks in den Zeilen **(02)** bis **(05)** die äußere Variable `ivar` der Zeile **(01)** überdeckt. Zugegeben, das Beispiel ist kein guter Stil, und die meisten Compiler geben hier auch eine Warnmeldung aus, aber es zeigt sehr schön die Sichtbarkeit von Variablen.

3.12 void – ein unvollständiger Typ

Nicht erwähnt wurde bisher `void` – ein leerer Datentyp, der keine Werte aufnehmen kann. `void` wird auch als unvollständiger Typ bezeichnet, weshalb auch keine Variable von diesem Typ erzeugt werden kann. Wird dieser Typ bei einer Funktion als Rückgabewert verwendet, so gibt die Funktion keinen Wert zurück. Auch bei den Zeigern gibt es eine Variable vom Typ `void`, mit der Sie auf Objekte mit unspezifiziertem Typ zeigen.

3.13 Kontrollfragen und Aufgaben

1. Welche grundlegenden Datentypen für Ganzzahlen gibt es?
2. Womit können Sie eine Integervariable ohne Vorzeichen vereinbaren?
3. Aufgrund verschiedener Datenmodelle ist die Breite von Datentypen implementierungsabhängig. Was können Sie tun, wenn Sie einen Ganzzahltypen mit fester Breite benötigen?
4. Nennen Sie den grundlegenden Datentypen, der für Zeichen verwendet wird.
5. Ermitteln Sie die Größe in Byte der Datentypen `int` und `long long` auf Ihrem System.
6. Womit können Sie die implementierungsabhängigen minimalen und maximalen Wertbereiche für Ganz- und Gleitkommazahlen ermitteln?
7. Was bewirkt das Schlüsselwort `const` vor einem Datentyp?

Kapitel 4

Rechnen mit C und Operatoren

Nachdem Sie die grundlegenden Datentypen von C kennen, erfahren Sie nun, wie Sie Werte mit `scanf` einlesen und mithilfe der arithmetischen Operatoren und Standardfunktionen der Headerdatei `<math.h>` Berechnungen durchführen können. Und wenn wir schon bei den Operatoren sind, sollen auch gleich die Bit-Operatoren und der Inkrement- bzw. Dekrement-Operator behandelt werden.

4.1 Werte formatiert einlesen mit scanf

Damit Sie auf den nächsten Seiten etwas mehr Praxisbeispiele erstellen können, werden Sie in Kürze das Nötigste zur Funktion `scanf` kennenlernen. Mit dieser Funktion können Sie formatiert von der Standardeingabe einlesen. Sie ist das Gegenstück zu `printf` (siehe Abschnitt 2.2). Die Standardeingabe (`stdin`), von der eingelesen wird, ist normalerweise die Tastatur. `scanf` ist ebenfalls in der Headerdatei `<stdio.h>` deklariert, weshalb Sie auch hier wieder diesen Header inkludieren müssen.

Die Funktion `scanf` gibt `EOF` (häufig als –1 implementiert) zurück, wenn ein Fehler vor der Konvertierung bei der Funktion `scanf` aufgetreten ist. Ansonsten gibt `scanf` die Anzahl der erfolgreich eingelesenen Eingabewerte zurück. Dies kann auch 0 sein, wenn das angegebene Umwandlungszeichen (oder auch der Platzhalter) mit dem `%`-Zeichen nicht mit dem Typ der Eingabe übereinstimmt. Mit dem Rückgabewert von `scanf` können Sie praktisch die Eingabe auf Korrektheit testen, was Sie eigentlich auch immer tun sollten.

Sie können fast überall dieselben Formatelemente mit den Prozentzeichen für die Basisdatentypen verwenden, die Sie von `printf` her kennen. Bei `double` müssen Sie beispielsweise `%lf` für `scanf` anstatt `%f` wie bei `printf`

verwenden. Allerdings erwartet die Funktion `scanf` im Gegensatz zu `printf` die Adresse der Variablen, weshalb folgender Funktionsaufruf zu einer Fehlermeldung führen würde:

```
scanf("%d", iVar);   // Fehler!!!
```

Die Adresse einer Variablen erhalten Sie mit dem Adressoperator `&`. Daher müssen Sie `scanf` wie folgt verwenden:

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

Sie können den Adressoperator natürlich auch bei der Ausgabe von `printf` nutzen, womit Sie die (Speicher-)Adresse einer Variablen ausgeben.

Hierzu ein Beispiel, das ausnahmsweise auf ein Konstrukt vorgreift, welches an dieser Stelle noch nicht behandelt wurde, nämlich eine `if`-Überprüfung. Die `if`-Anweisung selbst lernen Sie in Abschnitt 5.1 kennen. Da das Überprüfen der Eingabe des Anwenders ein für das Gesamtverständnis essenzieller Aspekt ist, ist dieser Vorgriff sicher sinnvoll und vertretbar.

```
00  // kap004/listing001.c
01  #include <stdio.h>

02  int main(void) {
03    int iVar = 0;
04    printf("Bitte eine Ganzzahl eingeben: ");
05    int check = scanf("%d", &iVar);
06    if( check == EOF ) {
07      printf("Fehler bei scanf...\n");
08      return 1;  // Programm beenden
09    }
10    printf("%d Wert(e) eingelesen; ", check);
11    printf("der eingegebene Wert lautet: %d\n", iVar);
12    printf("Die Adresse von iVar lautet: %p\n", &iVar);
13    return 0;
14  }
```

In der Zeile **(04)** werden Sie durch die Ausgabe nach einer Zahl gefragt. `scanf` wartet in der Zeile **(05)** auf die Eingabe einer Zahl, die Sie mit ↵ bestätigen müssen. In der Zeile **(06)** prüfen Sie, ob bei der Funktion `scanf` ein Fehler aufgetreten ist und brechen das Programm mit einer Fehlermeldung in den Zeilen **(07)** und **(08)** ab. Wenn Sie eine korrekte Ganzzahl eingegeben haben, wird in der Zeile **(10)** der Wert `check` gleich 1 sein, und in der Zeile **(11)** geben Sie dann den mit `scanf` eingegebenen Wert aus.

In der Zeile **(12)** wird außerdem nochmals demonstriert, wie Sie die Adresse einer Variablen mit `printf` ausgeben können. Als Umwandlungszeichen für eine Adresse müssen Sie `%p` verwenden. Das Programm bei der Ausführung:

```
Bitte eine Ganzzahl eingeben: 12345
1 Wert(e) eingelesen; der eingegebene Wert lautet: 12345
Die Adresse von iVar lautet: 000000000013ff04
```

Das Problem an diesem Beispiel ist allerdings noch, dass der zurückgegebene `EOF`-Fehler nicht aussagt, ob auch ein gültiger Integerwert eingelesen wurde.

Hierfür müssen Sie den Rückgabewert auf die Anzahl der erfolgreich eingelesenen Werte prüfen. Wenn in diesem Beispiel ein gültiger Integerwert (und beispielsweise kein Buchstabe) eingegeben wurde, dann ist der Wert von `check` gleich 1. Folglich würde hier folgende Überprüfung den Fall abdecken, dass kein Fehler vor der ersten Konvertierung mit `scanf` (`EOF`) aufgetreten ist und dass ein gültiger Wert eingegeben wurde. Dies wäre der Fall, wenn der Rückgabewert in `check` gleich 1 wäre. Eine solche etwas wasserdichtere Überprüfung von `scanf` würde daher wie folgt aussehen:

```
...
  int check = scanf("%d", &iVar);
  if( check == EOF ) {
    printf("Fehler bei scanf...\n");
    return 1;  // Programm beenden
  }
```

```
  if( check != 1 ) {  // Wert nicht 1, dann Fehler
    printf("Fehler bei der Eingabe\n");
    return 1;  // Programm beenden
  }
...
```

Sie könnten den Rückgabewert von `scanf` also auch in einer `if`-Anweisung verwenden, um zu prüfen, ob die Anzahl der erfolgreich eingelesenen Werte auch mit dem geforderten (und auch der Anzahl der) Umwandlungszeichen übereinstimmt. Es ist schließlich durchaus möglich, mehrere Werte gleichzeitig mit `scanf` einzulesen, wie folgendes Beispiel zeigt:

```
00  // kap004/listing002.c
01  #include <stdio.h>

02  int main(void) {
03    int iVar1 = 0, iVar2 = 0;
04    printf("Bitte zwei Ganzzahlen eingeben: ");
05    int check = scanf("%d %d", &iVar1, &iVar2);
06    if( check != 2 ) {
07      printf("Fehler: Zwei Ganzzahlen erwartet!\n");
08      return 1;  // Programm beenden
09    }
10    printf("%d Wert(e) eingelesen: ", check);
11    printf("%d und %d\n", iVar1, iVar2);
12    return 0;
13  }
```

Das Programm bei der Ausführung:

```
Bitte zwei Ganzzahlen eingeben: 12 d
Fehler: Zwei Ganzzahlen erwartet!
*** Process returned 1 ***

Bitte zwei Ganzzahlen eingeben: 12 33
2 Wert(e) eingelesen: 12 und 33
```

4.2 Operatoren im Allgemeinen

Operatoren dienen dazu, Werte und Variablen miteinander zu verknüpfen. Neben mathematischen Berechnungen rufen Sie mit Operatoren beispielsweise auch Funktionen auf oder führen logische Verknüpfungen zusammen. Der Teil, womit oder worauf die Operatoren angewendet werden, wird als *Operand* bezeichnet. Eine solche Verknüpfung aus Operand und Operator erzeugt einen Rückgabewert, der wiederum als Operand verwendet werden kann. Ein einfaches Beispiel hierzu:

```
z = x + y;
```

Hier werden die beiden Operanden x und y mit dem Additionsoperator + miteinander verknüpft. Das Ergebnis dieser Operation wird wiederum als Operand genutzt, um es mithilfe des Operators = dem Operanden z zuzuweisen. Eine solche Zusammenfassung von mehreren Operationen wird auch als *Ausdruck* bezeichnet.

In C lassen sich die Operatoren in folgende drei Gruppen einteilen:

- Unäre Operatoren: Die unären Operatoren sind einstellige Operatoren mit einem Operanden, die entweder links oder rechts von dem Operanden stehen. Ein Beispiel ist der Minusoperator, wenn dieser als Vorzeichen eines Operanden wirkt (etwa –100). Wenn Sie den Operator beispielsweise auf einen positiven Wert anwenden, ändern Sie das Vorzeichen des Wertes.
- Binäre Operatoren: Die binären Operatoren verwenden zwei Operatoren für ihre Verknüpfung. Ein einfaches Beispiel hierfür sind die arithmetischen Operatoren wie etwa der für eine Addition zweier Werte (100 + 100).
- Ternäre Operatoren: Hierbei handelt es sich um einen dreistelligen Operator, von dem es in C mit dem Bedingungsoperator ?: nur einen gibt.

Für jeden Operator gibt es eine *Rangordnung* bzw. *Priorität* (engl. *operator precedence*). Sinngemäß haben beispielsweise die arithmetischen Operatoren +, -, * oder / eine höhere Priorität als der Zuweisungsoperator =, weshalb auch immer erst die arithmetische Berechnung durchgeführt und dann das Ergebnis an die Variable zugewiesen wird.

Auch gilt hier die bekannte Punkt-vor-Strich-Regelung der Mathematik, bei welcher der Multiplikationsoperator eine höhere Priorität hat als der Additionsoperator. Ein Beispiel:

```
5 + 10 * 2
```

In diesem Fall wird zunächst 10 mit 2 multipliziert und dann zu 5 addiert; das Ergebnis lautet 25. Aber Sie können auch – wie in der Mathematik üblich – bei der Addition eine Klammerung verwenden, wenn Sie zuerst 5 und 10 addieren und dann mit 2 multiplizieren wollen, was 30 ergibt. Ein Beispiel:

```
(5 + 10) * 2
```

Diese Beschreibung hat Ihnen gezeigt, dass viele Abarbeitungsregeln von Operatoren in C genauso funktionieren wie im täglichen Gebrauch in mathematischen Berechnungen. Trotzdem gibt es in der Programmierung natürlich immer wieder einige Besonderheiten.

Wenn Sie Operatoren mit derselben Rangfolge verwenden, gibt es auch noch eine vorgeschriebene *Abarbeitungsrichtung* (auch: *Assoziativität*), mit der festgelegt wird, ob die Operatoren von links nach rechts (*linksassoziativ*) oder von rechts nach links (*rechtsassoziativ*) abgearbeitet werden. Wenn Sie beispielsweise zwei Multiplikationen wie folgt durchführen:

```
a * b * c
```

dann wird von links nach rechts gerechnet. Zunächst wird also `a` mit `b` multipliziert und dann das Ergebnis mit `c`. Verwenden Sie hingegen den Zuweisungsoperator, zum Beispiel

```
a = b = c
```

dann wird von rechts nach links gearbeitet. Zuerst wird der Wert von `c` an `b` und dann der Wert an `a` zugewiesen.

Überblick zur Rangordnung und Abarbeitungsrichtung

Einen Überblick zur Rangordnung und Abarbeitungsrichtung der einzelnen Operatoren finden Sie im Anhang A dieses Buchs.

Bei der Abarbeitung einzelner Operatoren eines Ausdrucks entstehen Zwischenergebnisse, die nach der Berechnung entfernt werden. Der Wert eines solchen Ausdrucks wird als *Rvalue* (manchmal übersetzt als R-Wert) bezeichnet. Ein Beispiel:

```
100 + 100;
```

Auch wenn ein solcher Ausdruck keinen wirklichen Effekt hat, da der Wert der Berechnung sofort wieder verworfen wird, ist es möglich, ihn in C so zu verwenden.

Werte hingegen, die in einem Speicher gespeichert sind, werden als *Lvalues* (manchmal übersetzt als L-Werte) bezeichnet. Solche Werte werden von Operatoren benötigt, die einen Wert speichern oder die Speicheradresse des Wertes verwenden. Ein Beispiel:

```
int a;
a = 100;  // lvalue = rvalue
```

Ein *Lvalue* ist daher immer ein Wert, der im Speicher lokalisierbar ist. Auch der Standard (seit C11) empfiehlt mittlerweile, sich das L in *Lvalue* als »lokalisierbar« (*locator value*) zu merken und sich den Begriff *Rvalue* nur noch als den Wert eines Ausdrucks (*value of an expression*) vorzustellen. So können Sie beispielsweise mit Operatoren einen Wert nur an lokalisierbare Werte (*Lvalues*) zuweisen. Daher ist Folgendes auch nicht möglich:

```
100 = a;  // rvalue = lvalue  -> Fehler!!!
```

Das Literal 100 ist hier nicht lokalisierbar, weshalb die Zuweisung nicht erlaubt ist. Somit muss beispielsweise links neben dem Zuweisungsoperator immer ein *Lvalue* stehen, der auch änderbar ist. Es ist ja auch möglich, Variablen mit `const` als nicht mehr änderbar zu qualifizieren.

Jetzt sind Sie zumindest ein wenig mit den Begriffen *Lvalues* und *Rvalues* vertraut, und wenn Sie einmal einen Fehler wie `Invalid lvalue` oder `lvalue required` vor sich haben, wissen Sie zumindest, was es damit auf sich hat.

4.3 Arithmetische Operatoren

Arithmetische Operatoren sind binäre Operatoren. Das heißt, der Operator hat immer zwei (*lateinisch* bi) Operanden (beispielsweise `[Operand][Operator][Operand]` oder einfach `10+10`). Die Tabelle 4.1 zeigt, welche arithmetischen Operatoren in C zur Verfügung stehen.

Operator	Bedeutung
+	Addiert zwei Werte.
-	Subtrahiert zwei Werte.
*	Multipliziert zwei Werte.
/	Dividiert zwei Werte. Eine Division durch 0 ist nicht erlaubt.
%	Rest einer Division (Modulo). Funktioniert nur mit Ganzzahlen als Datentyp. Kann nicht mit Fließkommazahlen verwendet werden. Außerdem darf niemals eine Division durch 0 mit dem Modulo-Operator vorgenommen werden.

Tabelle 4.1 Darstellung von arithmetischen Operatoren in C

Hierzu ein einfaches Beispiel mit dem Multiplikationsoperator, das die Fläche eines Kreises anhand des Radius berechnet ($A = r^2 \times PI$):

```
00 // kap004/listing003.c
01 #include <stdio.h>

02 int main(void) {
03   const double pi = 3.14159265358979;
04   double r = 0.0;
05   printf("Radius eingeben: ");
06   int check = scanf("%lf", &r);
07   if( check != 1 ) {
08     printf("Fehler beim Einlesen ...\n");
09     return 1;  // Programm beenden
10   }
```

```
11    double aKreis = r * r * pi;
12    printf("Kreisflaeche betraegt: %lf\n", aKreis);
13    return 0;
14  }
```

Als Gleitkommatyp wählen Sie `double` und definieren gleich den Wert von `pi` in Zeile **(03)** als Konstante. Nachdem Sie den Wert für den Radius `r` in Zeile **(06)** eingelesen und die Eingabe in den Zeilen **(07)** bis **(10)** überprüft haben, wird die Berechnung in Zeile **(11)** durchgeführt. Dort werden die einzelnen Werte multipliziert, und das Ergebnis wird an die Variable `aKreis` zugewiesen. In der Zeile **(12)** wird das Ergebnis dieser Berechnung ausgegeben. Das Programm bei der Ausführung:

```
Radius eingeben: 19.2
Kreisflaeche betraegt: 1158.116716
```

Auf diese Weise können Sie selbstverständlich auch Berechnungen mit dem Additionsoperator, Subtraktionsoperator, Divisionsoperator oder dem Modulo-Operator durchführen.

Erweiterte Darstellung arithmetischer Operatoren

Die arithmetischen Operatoren aus dem vorherigen Abschnitt lassen sich auch in einer kürzeren Schreibweise notieren, wie Tabelle 4.2 zeigt.

Operator	Bedeutung
`+=`	`Val1 += Val2` ist gleichwertig mit `Val1 = Val1 + Val2`
`-=`	`Val1 -= Val2` ist gleichwertig mit `Val1 = Val1 - Val2`
`*=`	`Val1 *= Val2` ist gleichwertig mit `Val1 = Val1 * Val2`
`/=`	`Val1 /= Val2` ist gleichwertig mit `Val1 = Val1 / Val2`
`%=`	`Val1 %= Val2` ist gleichwertig mit `Val1 = Val1 % Val2`

Tabelle 4.2 Erweiterte Darstellung von arithmetischen Operatoren in C

4.4 Inkrement- und Dekrement-Operator

Bei einem Inkrement oder Dekrement wird der Wert einer Variablen um 1 erhöht bzw. heruntergezählt. Diese Operatoren werden in C folgendermaßen geschrieben (siehe Tabelle 4.3).

Operator	Bedeutung
`++`	Inkrement-Operator (Variable wird um 1 erhöht)
`--`	Dekrement-Operator (Variable wird um 1 verringert)

Tabelle 4.3 Inkrement- und Dekrement-Operator

Für die Verwendung dieser beiden Operatoren, die sich neben Ganzzahlen auch auf Fließkommazahlen anwenden lassen, gibt es jeweils zwei Möglichkeiten (siehe Tabelle 4.4).

Anwendung	Bedeutung
`var++`	Postfix-Schreibweise, erhöht den Wert von `var`, gibt aber noch den alten Wert an den aktuellen Ausdruck weiter.
`++var`	Präfix-Schreibweise, erhöht den Wert von `var` und gibt diesen sofort an den aktuellen Ausdruck weiter.
`var--`	Postfix-Schreibweise, reduziert den Wert von `var`, gibt aber noch den alten Wert an den aktuellen Ausdruck weiter.
`--var`	Präfix-Schreibweise, reduziert den Wert von `var` und gibt diesen sofort an den aktuellen Ausdruck weiter.

Tabelle 4.4 Postfix- und Präfix-Schreibweisen

Anwendungsgebiet von Inkrement- und Dekrement-Operatoren

Hauptsächlich werden Inkrement- und Dekrement-Operator bei Schleifen verwendet. Sie sind beide unärer Natur und können nur auf veränderbare *Lvalues* angewendet werden.

Das folgende Beispiel demonstriert Ihnen den Inkrement-Operator (++) etwas genauer. Analog verhält sich natürlich auch der Dekrement-Operator (--).

```
00  // kap004/listing004.c
01  #include <stdio.h>

02  int main(void) {
03    int ival = 1;
04    printf("ival : %d\n", ival);
05    ival++;
06    printf("ival : %d\n", ival);
07    printf("ival : %d\n", ival++);
08    printf("ival : %d\n", ival);
09    printf("ival : %d\n", ++ival);
10    return 0;
11  }
```

Das Programm bei der Ausführung:

```
ival : 1
ival : 2
ival : 2
ival : 3
ival : 4
```

Bei der ersten Verwendung des Inkrement-Operators in der Zeile **(05)** wird der alte Wert noch an den aktuellen Ausdruck weitergegeben. Da hier die Inkrementierung für sich alleine steht, ist dies gleichzeitig auch der aktuelle Ausdruck. Zeile **(06)** hingegen ist der nächste Ausdruck, weshalb hier der Wert von `ival` 2 lautet. Sie werden das besser verstehen, wenn Sie die Zeile **(07)** ausführen. Hier lautet die Ausgabe nach wie vor 2, weil die Inkrementierung innerhalb des aktuellen Ausdrucks ausgeführt wird, und dieser endet am Semikolon, wo sich dann auch das späte Inkrement auswirkt. Daher hat erst der nächste Ausdruck in der Zeile **(08)** den erwarteten Wert 3. Wollen Sie den Wert einer Variablen sofort innerhalb eines Ausdrucks inkrementieren, dann müssen Sie statt der Postfix- die Präfix-Schreibweise verwenden, wie Sie in Zeile **(09)** sehen.

Nebeneffekte und Sequenzpunkte

An dieser Stelle soll anhand des Inkrementoperators ++ noch kurz auf die Begriffe Nebeneffekt und Sequenzpunkt eingegangen werden. Ein *Nebeneffekt* (oder auch Seiteneffekt bzw. engl. *side effect*) tritt auf, wenn der *Lvalue* nicht nur ausgewertet, sondern gleich auch verändert wird. Das folgende Beispiel zeigt, in welchem Zusammenhang die sogenannten Nebeneffekte stehen:

```
01  int iVar = 5;
02  int aVar = iVar + iVar++;
```

In diesem Beispiel wird an `aVar` der Rückgabewert der Berechnung zugewiesen. Je nachdem, wann der Nebeneffekt ausgewertet wird (ob `iVar` noch 5 oder 6 ist), gibt diese Addition 10 oder 11 zurück. Mit einem *Sequenzpunkt* (engl. *sequence point*) wird der Punkt festgelegt, bis zu dem der Nebeneffekt ausgewertet sein muss. Im Beispiel befindet sich der Sequenzpunkt beim Auftreten des Semikolons am Ende der Anweisung der Zeile **(02)**. Somit ist jedes Auftreten eines Semikolons ein Sequenzpunkt. Nach dem Sequenzpunkt besitzt die Variable `iVar` auf jeden Fall den Wert 6.

4.5 Bit-Operatoren

Für den direkten Zugriff auf die binäre Darstellung für Ganzzahlen können Sie auf die Bit-Operatoren zurückgreifen. Im Tabelle 4.5 finden Sie eine Übersicht, welche Bit-Operatoren es gibt. Alle stehen, wie schon die arithmetischen Operatoren, auch in einer erweiterten Schreibweise zur Verfügung.

Bit-Operator	Erweitert	Bedeutung
&	&=	bitweise UND-Verknüpfung (and)
|	|=	bitweise ODER-Verknüpfung (or)
^	^=	bitweises XOR

Tabelle 4.5 Übersicht über die bitweisen Operatoren

Bit-Operator	Erweitert	Bedeutung
~		bitweises Komplement
>>	>>=	Rechtsverschiebung
<<	<<=	Linksverschiebung

Tabelle 4.5 Übersicht über die bitweisen Operatoren (Forts.)

Bit-Operatoren und Fließkommazahlen

Die Operanden für die Verwendung mit Bit-Operatoren müssen immer ganzzahlige Datentypen sein. `float` oder `double` dürfen nicht als Operanden verwendet werden.

Bitweise Operatoren können nützlich sein, wenn Sie beispielsweise in einem Programm eine Zahl darauf testen wollen, ob ein bestimmtes Bit gesetzt ist, oder wenn Sie gezielt einzelne Bits setzen oder löschen möchten.

Bitweises UND

Ein bitweises UND wird durch das &-Zeichen repräsentiert, überprüft zwei Bitfolgen und führt eine logische UND-Verknüpfung durch. Das bedeutet, wenn beim Paar der Verknüpfung beide Bits 1 sind, ist das Ergebnis-Bit ebenfalls 1. Ansonsten ist das Ergebnis-Bit 0. Diese UND-Verknüpfung eignet sich relativ gut, um Bits in Bitmustern auszublenden. Ein Beispiel:

```
unsigned int val = 5;
unsigned int ret = val & 3;
```

Hier wurde Folgendes durchgeführt und auf die ersten vier Bits gekürzt:

```
   0101  (5)
&  0011  (3)
------------
   0001  (1)
```

Folgende Regeln gelten daher für den bitweisen UND-Operator (siehe Tabelle 4.6).

Bit A	Bit B	Ergebnis-Bit
0	0	0
0	1	0
1	0	0
1	1	1

Tabelle 4.6 Regeln für bitweise UND-Verknüpfung

Bitweises ODER

Wenn Sie den bitweisen-ODER-Operator `|` auf zwei gleich lange Bitfolgen anwenden, werden die Bitpaare zu einem logischen ODER verknüpft. Das bedeutet, dass wenn bei mindestens einem der beiden Bitpaare das Bit 1 ist, auch das Ergebnis 1 lautet. Damit ist der bitweise ODER-Operator das Gegenstück zum UND-Operator, mit dem Sie Bits in Bitmustern einblenden können. Das Beispiel `5|3` sieht wie mit diesem Operator wie folgt aus:

```
  0101  (5)
| 0011  (3)
------------
  0111  (7)
```

Folgende Regeln gelten daher für den bitweisen ODER-Operator (siehe Tabelle 4.7).

Bit A	Bit B	Ergebnis-Bit
0	0	0
0	1	1
1	0	1
1	1	1

Tabelle 4.7 Regeln für bitweise ODER-Verknüpfung

Bitweises XOR

Anders als der bitweise ODER-Operator liefert das bitweise XOR (oder auch exklusive ODER) als Endergebnis zweier Bitfolgen 1 zurück, wenn beide Bits unterschiedlich sind. Der exklusive ODER-Operator eignet sich daher sehr gut, um Bits zu invertieren, also zum Beispiel `5^3`:

```
   0101  (5)
^  0011  (3)
------------
   0110
```

Folgende Regeln gelten für den bitweisen XOR-Operator (siehe Tabelle 4.8).

Bit A	Bit B	Ergebnis-Bit
0	0	0
0	1	1
1	0	1
1	1	0

Tabelle 4.8 Regeln für bitweise XOR-Verknüpfung

Bitweises Komplement

Der NOT-Operator (~) wirkt sich auf Zahlen so aus, dass er jedes einzelne Bit invertiert. Bei vorzeichenbehafteten Datentypen entspricht das einer Negation mit anschließender Subtraktion von 1. Für diese NOT-Verknüpfung gilt folgende Tabelle 4.9.

BitA	~BitA
0	1
1	0

Tabelle 4.9 Regeln einer bitweisen NOT-Verknüpfung

Links- bzw. Rechtsverschiebung (Shift-Operatoren)

Mit einer Linksverschiebung (<<) bzw. Rechtsverschiebung (>>) werden alle Bits einer Zahl um n Stellen nach links gerückt. Die rechts oder links entstehenden Leerstellen werden mit 0 aufgefüllt, beispielsweise `3<<1`:

```
      0011   (3)
<<1   0110   (6)
```

Hier haben Sie eine Multiplikation mit 2 durchgeführt.

4.6 Implizite Typumwandlung

Wenn die Operanden eines Ausdrucks nicht vom selben Typ sind, kann der Compiler selbstständig eine implizite Typumwandlung durchführen. Diese automatische Typumwandlung läuft nach bestimmten Regeln ab, worauf in diesem Abschnitt kurz eingegangen werden soll.

Implizite Umwandlungen auch bei Funktionen

Gleiches gilt beim Aufruf einer Funktion, wenn das Argument nicht mit dem Typ des Parameters übereinstimmt. Hier führt der Compiler ebenfalls eine implizite Umwandlung des Typs durch. Dasselbe gilt für Rückgabewerte von Funktionen, wenn der rückzugebende Ausdruck vom Rückgabetyp unterschiedlich ist.

4.6.1 Arithmetische Umwandlung

Von einer *arithmetischen Typumwandlung* ist die Rede, wenn zwei unterschiedliche Typen so umgewandelt werden, dass sie im Idealfall den ursprünglichen Wert behalten. In der Praxis wird eine solche Umwandlung bei Operatoren verwendet, die zwei arithmetische Operanden erwarten. Ein einfaches Beispiel dazu:

```
01  int iVal = 1;
02  double dVal = 1.5;
03  printf("%f\n", iVal + dVal);
```

In diesem Beispiel wird in der Zeile **(03)** der Wert `iVal` vom Typ `int` in eine `double`-Zahl umgewandelt, weil dieser Typ »mächtiger« als der `int`-Typ ist. Sie können das gerne testen, indem Sie das Umwandlungszeichen der Zeile **(03)** auf `%d` (für `int`-Wert) ändern. Bei der Übersetzung sollte sich der Compiler mit einer Warnung bei Ihnen melden, dass der Formatstring von `printf` nicht mit dem Typ übereinstimmt.

Der Mächtigere bekommt den Zuschlag ...

Bei einer arithmetischen Umwandlung können Sie sich somit darauf verlassen, dass der schwächere in den mächtigeren Typ umgewandelt wird. Der mächtigste Typ von allen ist `long double`. Zwar gibt es hierbei abhängig vom Compiler noch spezielle Ausnahmefälle, auf die hier allerdings nicht näher eingegangen wird. Es sollte vielleicht noch angemerkt werden, dass nichts umgewandelt wird, wenn beide Operanden denselben Typ haben.

Das Ziel dieser Regeln bei einer arithmetischen Umwandlung ist es somit, dass im folgenden theoretischen Beispiel sowohl `Operand1` als auch `Operand2` auf denselben Typ gebracht werden:

```
Operand1 Operator Operand2
```

Die Gleitkommatypen `long double`, `double` und `float` sind mächtiger als die Integertypen. Daher gilt der Reihe nach:

- Ist ein Operand ein `long double`, wird auch der andere Operand in ein `long double` umgewandelt.
- Ist ein Operand ein `double`, wird auch der andere Operand in ein `double` umgewandelt.
- Ist ein Operand ein `float`, wird auch der andere Operand in ein `float` umgewandelt.

Wenn kein Operand ein Gleitkommatyp ist, werden diese Regeln auch in einer bestimmten Reihenfolge bei den Integertypen angewendet, obgleich es hier etwas schwieriger ist, eine einheitliche Regel wie bei den Gleitkommazahlen aufzulisten: Zwischenzeitlich sind ja mit C99 die

Typen `long long int` (höchste Mächtigkeit) und `_Bool` (niedrigste Mächtigkeit) hinzugekommen, und auch `signed` und `unsigned` wurden mittlerweile als gleich mächtig notiert. Daher können Sie sich hierbei zumindest an der folgenden grundlegenden Reihenfolge orientieren, die bei den meisten modernen Compilern (seit C11) funktionieren sollte:

- Wenn beide Operanden `signed`-Integertypen oder beide Operanden `unsigned`-Integertypen sind, wird der Operand mit einer niedrigeren Mächtigkeit in den Typ mit dem Operanden der höheren Mächtigkeit umgewandelt.
- Wenn der eine Operand ein `unsigned`-Integertyp ist, der andere ein `signed`-Integertyp mit gleicher oder niedrigere Mächtigkeit, wird der `signed`-Integertyp in den Operanden des `unsigned`-Integertyps umgewandelt.
- Wenn ein Operand ein `signed`-Integertyp ist, welcher die Werte des anderen Operanden vom `unsigned`-Integertyp abbilden kann, wird der `unsigned`-Integertyp in den Operanden des `signed`-Integertypen konvertiert.
- Ansonsten werden beide Operanden in einen `unsigned`-Integertypen umgewandelt, welcher dieselbe Mächtigkeit hat wie der entsprechende `signed`-Integertyp.

Es gibt geringfügige Unterschiede zwischen dem Regelwerk von C90 und C11, aber diese aufgelisteten Punkte dürften in der Reihenfolge recht gut die Integer-Umwandlungen beschreiben.

4.6.2 Typpromotionen

Zu diesem Regelwerk der arithmetischen Umwandlung kommen noch die *Typpromotionen*. Die Rede ist von einer Promotion (oder auch Typerweiterung), wenn der Umwandlungstyp den kompletten Umfang des umzuwandelnden Typen komplett abbilden kann, ohne dass es zu Verlusten kommt. Eine solche Promotion wird vom Compiler automatisch durchgeführt. So werden Integertypen bevorzugt in ein `int` und Gleitkommatypen in `double` umgewandelt, weil diese häufig als die effizientesten Typen gelten.

Trifft keine Regel der arithmetischen Umwandlung bei Integertypen zu und ist ein Operand beispielsweise vom Typ `signed char`, `signed short` oder

wchar_t, werden beide Operanden in ein signed int umgewandelt. Gleiches gilt, wenn ein Operand ein unsigned char oder unsigned short ist: Dann werden beide Operanden in ein unsigned int umgewandelt. Eine ähnliche Promotion gibt es auch bei den Gleitkommazahlen bei der Umwandlung vom Typ float nach double.

4.6.3 Was nicht geht!

Bei den arithmetischen Umwandlungen und Promotionen gibt es auch Umwandlungen, die nicht verlustfrei durchgeführt werden können. So ist es natürlich nicht möglich, die Nachkommastelle einer Gleitkommazahl in einem Integertyp zu speichern. Ebenso ist es nicht möglich, einen negativen signed-Typen in einem vorzeichenlosen unsigned-Typen zu speichern. Und auch wenn es logisch erscheint, können Sie niemals den Wert eines breiteren Typs in einem kleineren Typ unterbringen, wenn der breitere Typ bereits den Wertebereich des kleineren Typs sprengt.

4.7 Explizites Casting von Typen

Wollen Sie eine implizite Umwandlung des Compilers vermeiden, können Sie mithilfe des *cast*-Operators die Typenumwandlung selbst vornehmen. Eine solche explizite Anweisung, um dem Compiler die Umwandlung der Daten mitzuteilen, können Sie mit dem *cast*-Operator wie folgt festlegen:

```
(Typ) Ausdruck
```

Damit weisen Sie den Compiler an, den Wert des Ausdrucks in den Typ umzuwandeln, der zwischen den Klammern angegeben ist. Zunächst ein Beispiel ohne den *cast*-Operator:

```
01  int iVal1 = 10, iVal2 = 3;
02  double dVal = iVal1 / iVal2;
03  printf("%lf\n", dVal);  // = 3.000…
```

Hier wird eine Ganzzahldivision von 10 durch 3 durchgeführt und dann an eine double-Variable zugewiesen. Da zwei int-Werte für die Division verwendet wurden, kann logischerweise kein Wert nach dem Komma dar-

gestellt werden. Wäre nur einer der beiden Werte eine Gleitpunktzahl, würde intern eine übliche arithmetische Umwandlung durchgeführt. In diesem Beispiel kommt es daher zu einem Datenverlust.

In solch einem Fall können Sie sich mit einer expliziten Umwandlung mit dem *Cast-Operator* helfen. Damit schreiben Sie dem Compiler vor, eine Typenumwandlung durchzuführen. Bezogen auf unser Beispiel könnten Sie den Typen wie folgt explizit umwandeln:

```
01  int iVal1 = 10, iVal2 = 3;
02  double dVal = (double)iVal1 / (double)iVal2;
03  printf("%lf\n", dVal);  // = 3.3333…
```

Der Wert von `iVal1` und `iVal2` wird jeweils explizit in einen `double`-Typ konvertiert, und die Berechnung wird an die double-Variable zugewiesen. Die Umwandlung gilt allerdings nur während dieser Ausführung. `iVal1` und `iVal2` bleiben nach wie vor vom Typ `int`.

In dem Beispiel hätte es auch ausgereicht, nur einen der beiden Typen zu *casten*. Dank der üblichen arithmetischen Umwandlung wird aber auch der andere Typ automatisch umgewandelt:

```
01  int iVal1 = 10, iVal2 = 3;
02  double dVal = (double)iVal1 / iVal2;
03  printf("%lf\n", dVal);  // = 3.3333…
```

Allerdings ist es nicht möglich, jeden beliebigen Typen in einen anderen umzuwandeln, und einige Umwandlungen können auch unsicher und fehleranfällig sein, wenn dabei Zeiger im Spiel sind. Abhängig von der Warneinstellung geben die meisten Compiler allerdings auch eine Warnmeldung aus, wenn versucht wird, etwas umzuwandeln, was entweder nicht möglich ist oder als problematisch gilt.

4.8 Mathematische Funktionen in C

Die Standardbibliothek beinhaltet eine umfangreiche Sammlung von mathematischen Funktionen, zu denen Sie hier einen kleinen Überblick erhalten. Die meisten dieser Funktionen sind in der Headerdatei `<math.h>`

deklariert. Sie sind zu einem großen Teil für Gleitpunktzahlen oder für komplexe Gleitpunkttypen (aus der Headerdatei `<complex.h>`) geeignet. Zwar bietet die Standardbibliothek auch einige Funktionen für ganzzahlige Typen; diese sind aber vorwiegend in der Headerdatei `<stdlib.h>` bzw. für den Typ `intmax_t` in `<inttypes.h>` deklariert. Des Weiteren sind in der Headerdatei `<tgmath.h>` typengenerische Makros definiert, mit denen es möglich ist, mathematische Funktionen mit einem einheitlichen Namen aufzurufen, und zwar unabhängig vom Typ des Argumentes.

Mathe mit Linux

Damit Sie auch bei Linux-Programmen die mathematische Standardbibliothek verwenden können, müssen Sie den Compiler-Flag `-lm` (beispielsweise `gcc -o programm programm.c` **`-lm`**) hinzulinken.

Zu jeder mathematischen Funktion gibt es eine Version mit `float` bzw. `float _Complex`, `double` bzw. `double _Complex` und eine Version für `long double` bzw. `long double _Complex`. Die Versionen von `float` bzw. `float _Complex` haben das Suffix `f` am Ende des Funktionsnamens, die Versionen für `long double` bzw. `long double _Complex` das Suffix `l`. Für die Version von `double` bzw. `double _Complex` wird kein Suffix benötigt. Sofern Sie allerdings die Headerdatei `<tgmath.h>` verwenden, können Sie das Suffix ganz außer Acht lassen.

<complex.h>: no such file or directory

`<complex.h>` wurde erst mit dem C99-Standard eingeführt. Der Compiler unterstützt die komplexen Standardfunktionen also nur, wenn er C99-konform ist.

Wenn Sie beispielsweise die Funktion zum Ziehen der Quadratwurzel für reelle `double`-Zahlen verwenden wollen:

```
double sqrt(double zahl);
```

dann existieren für die Funktion noch die `float`- und die `long double`-Versionen:

```
float sqrtf(float zahl);
long double sqrtl(long double zahl);
```

Gleiches gilt auch für die aufgelistete komplexe Gleitpunkttyp-Version. Diese beginnt zusätzlich mit dem Präfix `c`:

```
double complex csqrt(double complex z);
```

Auch von dieser Version gibt es zwei weitere Versionen:

```
float complex csqrtf(float complex z);
long double complex csqrtl(long double complex z);
```

Ein einfaches Anwendungsbeispiel demonstriert die Verwendung zum Ziehen der Quadratwurzel mit `sqrt` aus `<math.h>`. Hierbei soll jeweils einmal der Fließkommatyp `float`, `double` und `long double` verwendet werden.

```
00 // kap004/listing005.c
01 #include <stdio.h>
02 #include <stdlib.h>
03 #include <math.h>
04 #include <complex.h> // C99

05 int main(void) {
06   long double ldval=8.8;
07   double dval=5.5;
08   float  fval=3.3;

09   // Quadratwurzel mit reellen Zahlen
10   printf("Quadratwurzel-Berechnungen:\n");
11   printf("(long double) sqrtl(%Lf) = %Lf\n",
                          ldval, sqrtl(ldval));
12   printf("(double) sqrt(%lf) = %lf\n",dval,sqrt(dval));
13   printf("(float) sqrtf(%f) = %f\n",fval,sqrtf(fval));
14   // Berechnung mit komplexen Zahlen
15  double pi = 4 * atan(1.0);
16  double complex c = cexp(I * pi);
```

```
17   printf("%lf + %lf * i\n", creal(c), cimag(c));
18   return 0;
19 }
```

In den Zeilen **(11)** bis **(13)** werden jeweils Quadratwurzeln von `long double`, `double` und einer `float`-Zahl mit der entsprechenden Version des Fließkommatyps gezogen. In den Zeilen **(15)** bis **(17)** wird die Verwendung von Standardfunktionen für komplexe Zahlen (seit C99 dabei) demonstriert. Achten Sie außerdem bei der formatierten Ausgabe auf die richtige Formatangabe (`%f`, `%lf` und `%Lf`) des entsprechenden Gleitpunkttyps.

Typengenerische Makros <tgmath.h>

`<tgmath.h>` wurde mit dem C99-Standard eingeführt. In `<tgmath.h>` sind die Headerdateien `<math.h>` und `<complex.h>` inkludiert und die typengenerischen Makros definiert. Der Vorteil dieser Makros liegt darin, dass Sie diese unabhängig vom Typ des Arguments die mathematischen Funktionen mit demselben Namen aufrufen können. Das bedeutet, Sie können außer Acht lassen, welche mathematischen Funktionen Sie für die Typen `float`, `double`, `long double`, `float complex`, `double complex` und `long double complex` aufrufen.

Wollen Sie beispielsweise eine Funktion zum Ziehen der Quadratwurzel verwenden, müssen Sie abhängig vom Datentyp zwischen sechs verschiedenen Varianten mit `sqrtf()`, `sqrt()`, `sqrtl()`, `csqrtf()`, `csqrt()` und `csqrtl()` unterscheiden.

Mit den typengenerischen Makros in `<tgmath.h>` brauchen Sie sich darum keine Gedanken mehr zu machen. Hier müssen Sie lediglich die Funktionen der `double`- bzw. `double complex`-Variante kennen, und ein Aufruf von `sqrt()` führt automatisch die entsprechende Erweiterung aus. Rufen Sie beispielsweise `sqrt()` mit einem `float complex`-Argument auf, wird automatisch die Erweiterung `csqrtf()` ausgeführt.

Hierzu folgt ein Beispiel, das diese typengenerischen Makros demonstrieren soll. Für alle reellen und komplexen Gleitpunkttypen wird immer nur die Funktion `sqrt()` zum Ziehen der Quadratwurzel aufgerufen. Das wäre ohne die typengenerischen Makros nicht denkbar, und bei Compilern, die

den C99-Standard nicht unterstützen, ist es auch nicht möglich. Hier sehen Sie das Listing:

```
00 // kap004/listing006.c
01 #include <stdio.h>
02 #include <stdlib.h>
03 #include <tgmath.h>

04 int main(void) {
05   long double ldval=8.8;
06   double dval=5.5;
07   float  fval=3.3;
08   double complex dcval= 1.0 + 2.0*I;

09   // Quadratwurzel mit reellen Zahlen
10   printf("Quadratwurzel-Berechnungen:\n");
11   printf("(long double) sqrt(%Lf) = %Lf\n"
                        ,ldval, sqrt(ldval));
12   printf("(double) sqrt(%lf) = %lf\n",dval, sqrt(dval));
13   printf("(float) sqrt(%f) = %f\n",fval, sqrt(fval));
14   double complex dcval_G = sqrt(dcval);
15   printf("(double complex) sqrt(4.0 + 2.0*I)"
       " = %lf + %lfi\n", creal(dcval_G), cimag(dcval_G));
16   return 0;
17 }
```

Hier finden Sie die Berechnungen mit der Funktion `sqrt`, ohne auf die Fließkommatypen zu achten, in den Zeilen **(11)** bis **(14)**.

Weitere mathematische Angebote der Standardbibliothek

Natürlich bietet die Standardbibliothek noch weitere nützliche Features rund um die mathematischen Funktionen. Dieser Bereich wurde besonders seit dem C99-Standard stark erweitert. Zu erwähnen wären hier Konstanten und Makros, um Fließkommawerte zu klassifizieren, sowie Makros für den Vergleich von reellen Zahlen und den Zugriff auf eine Fließkomma-Umgebung in `<fenv.h>`.

4.9 Kontrollfragen und Aufgaben

1. Welcher Fehler wurde hier gemacht? Bringen Sie das Beispiel zur Ausführung!

```
// kap004/aufgabe001.c
#include <stdio.h>

int main(void) {
  int iVar = 0;
  printf("Bitte eine Ganzzahl eingeben: ");
  int check = scanf("%d", iVar);

  if( check != 1 ) {
    printf("Fehler bei scanf ...\n");
    return 1;  // Programm beenden
  }
  printf("%d Wert(e) eingelesen; ", check);
  printf("der eingegebene Wert lautet: %d\n", iVar);
  return 0;
}
```

2. Schreiben Sie ein Listing, das nach einer Temperatur in Grad Celsius fragt. Diesen eingelesenen Wert rechnen Sie dann in Kelvin und Grad Fahrenheit um. Verwenden Sie `double` als Basisdatentyp. Die Formel, um aus einer Temperatur in Grad Celsius (TC) einen Wert in Grad Fahrenheit (TF) zu erhalten, lautet:

   TF = ((TC × 9) : 5) + 32

   Noch einfacher geht die Umrechnung von Celsius (TC) nach Kelvin (TK):

   TK = TC + 273,15

3. Was wird in den folgenden Zeilen für ein Wert ausgegeben?

```
int i = 1;
printf("i = %d\n", i--);
printf("i = %d\n", ++i);
printf("i = %d\n", i++);
printf("i = %d\n", ++i);
```

4. Was ist eine *implizite Umwandlung*?
5. Was ist eine *explizite Umwandlung,* und wann sollten Sie diese gegenüber der *impliziten Umwandlung* bevorzugen?
6. Womit wird eine *explizite Umwandlung* durchgeführt?

Kapitel 5

Bedingte Anweisung und Verzweigung

In diesem Kapitel erfahren Sie, was bedingte Anweisungen und Verzweigungen sind. Außerdem lernen Sie den einzigen ternären Bedingungsoperator (`?:`) und die logischen *Nicht*-, *Oder*- und *Und*-Operatoren kennen.

5.1 Bedingte Anweisung

Eine *Bedingte Anweisung* dient dazu, einen bestimmten Codeabschnitt nur dann auszuführen, wenn eine bestimmte Bedingung ausgeführt wird bzw. vorliegt.

Mini-Exkurs: Anweisungsblock

Wenn Sie bedingte Anweisungen oder Verzweigungen erstellen, werden Sie häufig aufgrund einer Bedingung oder Verzweigung mehr als nur eine Anweisung ausführen wollen. Mehrere Anweisungen können Sie in C in einem Anweisungsblock (engl. *compound statement*) zwischen geschweiften Klammern in einer Sequenz von Anweisungen zusammenfassen, beispielsweise:

```
{    // Anweisungsblock - Anfang
   Anweisung1;
   Anweisung2;
   ...
   AnweisungN;
}    // Anweisungsblock - Ende
```

Solche Anweisungsblöcke lassen sich auch ineinander verschachteln. Es empfiehlt sich jedoch, hiervon selten Gebrauch zu machen, weil sonst die Strukturierung und somit die Lesbarkeit des Programms erheblich leidet.

Bedingte Anweisung mit if

Die Syntax einer bedingten `if`-Anweisung in C sieht wie folgt aus:

```
if(ausdruck) {
   anweisung1;
}
anweisung2;
```

Zuerst wird die Bedingung `ausdruck` ausgewertet. Je nachdem, ob `ausdruck` wahr (ungleich 0) ist, wird die Anweisung `anweisung1` im Anweisungsblock ausgeführt. Anschließend wird die Programmausführung mit der Anweisung `anweisung2` fortgesetzt. Ist die Bedingung `ausdruck` allerdings unwahr (also gleich 0), werden die Anweisungen im Anweisungsblock nicht ausgeführt, und das Programm fährt sofort mit der Anweisung `anweisung2` fort.

Logischer Ausdruck

Die runden Klammern hinter `if` sind für den logischen Ausdruck unbedingt erforderlich. »Logisch« bedeutet in C immer ganzzahlig. Daher kann der Ausdruck in `if` jeden beliebigen numerischen Wert annehmen. 0 wird, wie bereits erwähnt, als falsch (unwahr) und jeder andere Wert als richtig (wahr) interpretiert.

Abbildung 5.1 stellt diese bedingte `if`-Anweisung in einem Programmablaufplan schematisch dar.

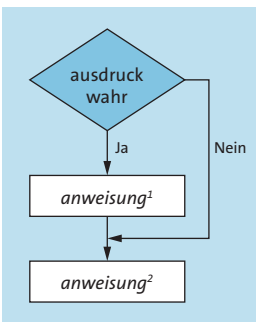

Abbildung 5.1 Programmablaufplan einer bedingten »if«-Anweisung

Sehen Sie sich hierzu folgendes Programmbeispiel an:

```
00 // kap005/listing001.c
01 #include <stdio.h>

02 int main(void) {
03   int ival = 0;
04   printf("Bitte eine Ganzzahl eingeben: ");
05   int check = scanf("%d", &ival);
06   if( check ) {
07     // Der Code wird ausgeführt, wenn die Bedingung, dass
08     // der Wert check nicht 0 ist, wahr ist.
09      printf("Ihre Eingabe: %d\n", ival);
10   }
11   printf("Außerhalb der if-Verzweigung\n");
12   return 0;
13 }
```

Das Programm ist absichtlich sehr einfach gehalten. In Zeile **(04)** werden Sie aufgefordert, eine Ganzzahl einzugeben. Die Eingabe wird mit `scanf` in der Zeile **(05)** eingelesen und in der Variablen `ival` (Zeile **(03)**) gespeichert. Der Rückgabewert von `scanf` wird in der Variablen `check` gespeichert. In Zeile **(06)** wird die Bedingung (oder auch der Ausdruck) daraufhin überprüft, ob die Anzahl der eingegebenen Werte von `scanf` »stimmt«. Dies ist der Fall, wenn `check` hier ungleich 0 (=wahr) ist. Hat der Anwender einen zum Umwandlungszeichen `%d` gültigen Wert eingegeben, wird die `printf`-Anweisung der Zeile **(09)** im Anweisungsblock der `if`-Anweisung ausgeführt. Es werden also alle Anweisungen im Anweisungsblock zwischen den Zeilen **(06)** bis **(10)** ausgeführt (hier allerdings nur eine einzige Anweisung).

Hat der Anwender hingegen einen Buchstaben wie a eingegeben, werden die Anweisungen zwischen dem Anweisungsblock (Zeilen **(06)** bis **(10)**) nicht ausgeführt, weil die `if`-Bedingung in diesem Fall 0 war, und es wird gleich mit der `printf`-Anweisung der Zeile **(11)** fortgefahren. Diese wird natürlich auch ausgeführt, wenn die Anweisungen im `if`-Anweisungsblock zwischen den Zeilen **(07)** bis **(10)** ausgeführt wurden.

Das Programm bei der Ausführung:

```
Bitte eine Ganzzahl eingeben: 10
Ihre Eingabe: 10
Außerhalb der if-Verzweigung

Bitte eine Ganzzahl eingeben: a
Außerhalb der if-Verzweigung
```

5.1.1 Vergleichsoperatoren

Ein Vergleich wie in der Zeile **(06)** des obigen Listings *listing001.c* dürfte Sie vielleicht ein wenig irritieren. Sie hätten hier genauso gut Folgendes verwenden können:

```
06    if(check !=0) {
07      // Der Code wird ausgeführt, wenn die Bedingung, dass
08      // der Wert check nicht 0 ist, wahr ist.
09       printf("Ihre Eingabe: %d\n", ival);
10    }
```

Dieser Vergleich in Zeile **(06)** mit dem `!=`-Operator (*Nicht-gleich*-Operator, einem Vergleichsoperator), ob der Ausdruck zwischen den Klammern von `if` ungleich 0 ist, entspricht dem in Listing (*listing001.c*) verwendeten Ausdruck. Bevor Sie weitere Verzweigungsmöglichkeiten kennenlernen, soll hier kurz auf die vorhandenen Vergleichsoperatoren eingegangen werden. Alle in Tabelle 5.1 aufgelisteten Vergleichsoperatoren vergleichen zwei Operanden und liefern einen Wert vom Typ `int` zurück. Ist der Vergleich wahr, geben die Operatoren einen Wert ungleich 0 zurück. Ist der Vergleich unwahr, wird 0 zurückgegeben.

Operator	Bedeutung	Beispiel	Rückgabewert
<	kleiner	a < b	Ungleich 0, wenn a kleiner als b, ansonsten 0

Tabelle 5.1 Übersicht über Vergleichsoperatoren

Operator	Bedeutung	Beispiel	Rückgabewert
<=	kleiner oder gleich	a <= b	Ungleich 0, wenn a kleiner oder gleich b, ansonsten 0
>	größer	a > b	Ungleich 0, wenn a größer b, ansonsten 0
>=	größer oder gleich	a >= b	Ungleich 0, wenn a größer oder gleich b, ansonsten 0
==	gleich	a == b	Ungleich 0, wenn a gleich b, ansonsten 0
!=	ungleich	a != y	Ungleich 0, wenn a ungleich b, ansonsten 0

Tabelle 5.1 Übersicht über Vergleichsoperatoren (Forts.)

So ganz kann man das Beispiel in *listing001.c* allerdings mit dem Vergleich auf ungleich 0 nicht stehen lassen, da `scanf` ja auch den Wert `EOF` im Fehlerfall vor der ersten Umwandlung zurückgeben kann, was oftmals als –1 implementiert ist. Somit würde der Anweisungsblock der bedingten `if`-Anweisung auch ausgeführt, wenn ein `EOF`-Fehler bei `scanf` aufgetreten wäre. Daher sind Sie bei diesem Beispiel auf der sicheren Seite, wenn Sie prüfen, ob `check` gleich dem Wert 1 entspricht. `scanf` gibt ja die Anzahl der erfolgreich eingelesenen Werte zurück (hier nur einen Wert). Daher sollten Sie das Beispiel besser wie folgt notieren:

```
06    if(check ==1 ) {
07      // Der Code wird ausgeführt, wenn die Bedingung, dass
08      // der Wert check gleich 1 ist, wahr ist.
09       printf("Ihre Eingabe: %d\n", ival);
10    }
```

Anstelle einer Zuweisung des Rückgabewertes von `scanf` an die zusätzliche Variable `check` können Sie den Rückgabewert dieser Funktion mit der Anzahl erfolgreich eingelesener Werte (hier: ein Wert) natürlich auch

direkt und ohne Umweg in die bedingte `if`-Anweisung einbauen. Zwar wurden Rückgabewerte von Funktionen noch nicht behandelt, aber das Beispiel dazu will ich Ihnen nicht vorenthalten:

```
...
  int ival = 0;
  printf("Bitte eine Ganzzahl eingeben: ");b
  if( scanf("%d", &ival) == 1 ) {
    // Der Code wird ausgeführt, wenn die Bedingung, dass
    // scanf den Wert 1 zurückgibt, wahr ist.
     printf("Ihre Eingabe: %d\n", ival);
  }
...
```

Vergleichsoperatoren müssen übrigens nicht unbedingt zwischen Vergleichen von `if`-Verzweigungen oder Schleifen stehen. So können Vergleichsoperatoren auch wie folgt verwendet werden:

```
int ival1 = 10 > 5;          // ival1 = 1
int ival2 = 5 == 4;          // ival2 = 0
int ival3 = ival1 != ival2;  // ival3 = 1
printf("%d : %d : %d\n", ival1, ival2, ival3);
```

5.2 Alternative Verzweigung

Mit einer Verzweigung können Sie festlegen, dass ein Programm in mehreren Abschnitten, die wiederum abhängig von einer Bedingung sind, ausgeführt wird. Damit können Sie im Programm auf unterschiedliche Zustände reagieren.

In der Praxis folgt häufig nach einer bedingten `if`-Anweisungen eine optionale und alternative Verzweigung. Sie wird auf jeden Fall ausgeführt, wenn die `if`-Bedingung nicht erfüllt, also 0 zurückgegeben wird. Realisiert wird die `else`-Verzweigung folgendermaßen:

```
if(ausdruck) {
   anweisung1;
}
```

```
else {
   anweisung2;
}
anweisung3;
```

Hierbei wird ebenfalls zuerst die Bedingung `ausdruck` ausgewertet. Je nachdem, ob `ausdruck` wahr (ungleich 0) ist, wird die Anweisung `anweisung1` im Anweisungsblock ausgeführt. Anschließend wird die Programmausführung mit der Anweisung `anweisung3` fortgesetzt. Ist die Bedingung `ausdruck` allerdings unwahr (also gleich 0), wird die Anweisung `anweisung2` im alternativen `else`-Anweisungsblock ausgeführt. Anschließend fährt das Programm mit der Anweisung `anweisung3` fort.

Kein »else« ohne »if«

Eine `else`-Alternative kann nur einer vorausgehenden `if`- oder `else-if`-Verzweigung folgen.

Abbildung 5.2 stellt diese `if`-Anweisung mit einer alternativen `else`-Verzweigung in einem Programmablaufplan schematisch dar.

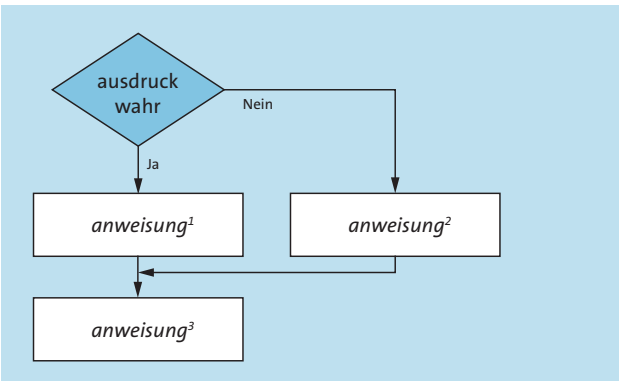

Abbildung 5.2 Programmablaufplan einer »if«-Anweisung mit alternativer »else«-Verzweigung

Hierzu soll das Listing *listing001.c* um eine alternative `else`-Verzweigung erweitert werden:

```
00  // kap005/listing002.c
01  #include <stdio.h>

02  int main(void) {
03    int ival = 0;
04    printf("Bitte eine Ganzzahl eingeben: ");
05    int check = scanf("%d", &ival);
06    if( check == 1 ) {
07      printf("Ihre Eingabe: %d\n", ival);
08    }
09    else {
10      printf("Fehler bei der Eingabe!\n");
11    }
12    printf("Außerhalb der if-Verzweigung\n");
13    return 0;
14  }
```

In diesem Beispiel wird in der Zeile **(06)** geprüft, ob der Rückgabewert von `check` gleich 1 ist und ob der eine geforderte Wert erfolgreich von `scanf` eingelesen werden konnte. Ist dies der Fall, wird der Wert ausgegeben, und das Programm fährt in der Zeile **(12)** mit der Ausführung fort. Ist der Wert von `check` nicht gleich 1, wurde entweder kein gültiger Wert passend zum Umwandlungszeichen eingegeben, oder es ist ein anderer Fehler bei `scanf` aufgetreten, und es wurde `EOF` zurückgegeben. In beiden Fällen ist die `if`-Bedingung 0 und unwahr, weshalb die Ausführung des Programms in den `else`-Block in der Zeile **(09)** bis **(11)** verzweigt. Dort wird eine `printf`-Anweisung mit dem Hinweis ausgegeben, dass ein Fehler bei der Eingabe gemacht wurde oder aufgetreten ist.

Das Programm bei der Ausführung:

```
Bitte eine Ganzzahl eingeben: 1234
Ihre Eingabe: 1234
Außerhalb der if-Verzweigung
```

```
Bitte eine Ganzzahl eingeben: x
Fehler bei der Eingabe!
Außerhalb der if-Verzweigung
```

5.3 Der Bedingungsoperator ?:

Der Bedingungsoperator `?:` ist der einzige dreiwertige Operator in C. Er ist auch als bedingte Bewertung bekannt. Im Prinzip handelt es sich bei diesem Operator um eine Kurzform der `if`- mit einer alternativen `else`-Anweisung. Die Syntax des Operators sieht wie folgt aus:

```
Bedingung ? Ausdruck1 : Ausdruck2
```

Ist die `Bedingung` wahr und gibt ungleich 0 zurück, wird $\texttt{Ausdruck}^1$ ausgeführt. Ist die `Bedingung` hingegen unwahr und gleich 0, wird $\texttt{Ausdruck}^2$ ausgeführt. Der Programmablauf ist somit derselbe wie bei einer bedingten `if`-Anweisung mit einer alternativen `else`-Verzweigung.

Dieser Operator sollte auf keinen Fall die `if`-Anweisung mit der alternativen `else`-Verzweigung ersetzen. Diese ist nach wie vor häufig besser lesbar als eine bedingte Auswertung mit dem ternären Operator. Trotzdem gibt es in der Praxis einfache Beispiele, bei denen der Bedingungsoperator einer `if`-Bedingung mit `else`-Verzweigung vorzuziehen ist. Mit dem folgenden Beispiel etwa soll der maximale oder der minimale Wert ermittelt werden und einer konstanten Variablen zugewiesen werden:

```
00  // kap005/listing003.c
01  #include <stdio.h>

02  int main(void) {
03    int val1 = 0, val2 = 0;
04    printf("Bitte zwei Ganzzahlwerte eingeben: ");
05    int check = scanf("%d %d", &val1, &val2);
06    if(check != 2) {
07      printf("Fehler bei der Eingabe ...\n");
08      return 1;
09    }
10    const int max = (val1 > val2) ?val1 :val2;
```

```
11    printf("Höherer Wert: %d\n", max);
12    return 0;
13  }
```

Das Hauptaugenmerk sollten Sie auf die Zeile **(10)** legen; hier wird der ternäre Operator verwendet. Die Auswertung des ternären Operators wird an die konstante Variable `max` übergeben. Es wird zunächst die Bedingung `(val1 > val2)` ausgewertet, also ob der eingegebene Ganzzahlwert von `val1` größer als `val2` ist. Trifft dies zu, wird die erste Anweisung hinter dem Fragezeichen ausgeführt. Im Beispiel wird nur der Wert der Variablen `val1` als Ausdruck verwendet und somit an die Variable `max` zugewiesen. Ist die Bedingung `(val1 > val2)` hingegen falsch, wird der zweite Ausdruck hinter dem Doppelpunkt ausgeführt. Dadurch wird der Wert der Variablen `val2` als Ausdruck verwendet und der konstanten Variablen `max` zugewiesen.

Theoretisch ist es natürlich auch möglich, die einzelnen Ausdrücke ineinander zu verschachteln. Der Lesbarkeit des Codes zuliebe kann ich Ihnen aber von solchen wilden Verschachtelungen nur abraten. Ein abschreckendes Beispiel:

```
big = (a>b) ?((a>c) ?a :c) :((b>c) ?b :c);
```

Diese Zeile macht nichts anderes, als den größten Wert der drei Variablen `a`, `b` und `c` zu ermitteln und an die Variable `big` zu übergeben.

5.4 Mehrfache Verzweigung mit if und else if

Reicht Ihnen eine bedingte `if`-Anweisung nicht aus, können Sie mehrere bedingte `if`-Anweisungen hintereinander verwenden. Dazu fügen Sie nach der ersten bedingten `if`-Anweisung der Reihe nach weitere bedingte `else-if`-Anweisungen an. Die Syntax sieht dann folgendermaßen aus:

```
if(ausdruck1) {
   anweisung1;
}
else if (ausdruck2) {
   anweisung2;
```

```
}
//   Weitere else-if-Anweisungen möglich
anweisung3;
```

Zuerst wird die Bedingung `ausdruck1` ausgewertet. Je nachdem, ob `ausdruck1` wahr (ungleich 0) ist, wird die Anweisung `anweisung1` im Anweisungsblock ausgeführt. Anschließend wird die Programmausführung mit der Anweisung `anweisung3` fortgesetzt. Ist die Bedingung `ausdruck1` allerdings unwahr (also gleich 0), fährt das Programm der Reihe nach mit der nächsten Überprüfung der Bedingung `ausdruck2` fort. Ist die Bedingung in `ausdruck2` wahr (ungleich 0), wird die Anweisung `anweisung2` ausgeführt. Anschließend fährt die Ausführung des Programms mit der Anweisung `anweisung3` fort.

Abbildung 5.3 stellt diese zusätzliche `else-if`-Anweisung in einem Programmablaufplan schematisch dar.

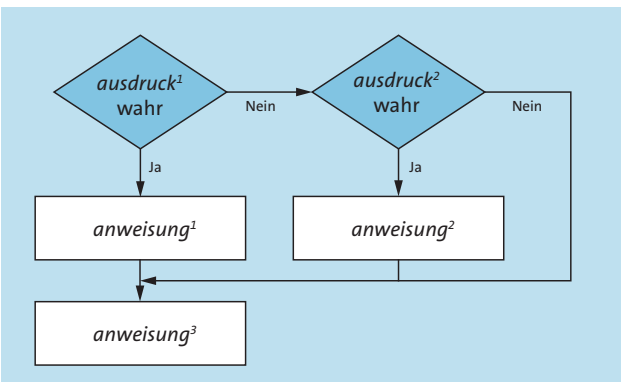

Abbildung 5.3 Programmablaufplan mit einer »else if«-Verzweigung

Natürlich ist es auch möglich, mehrere bedingte `else-if`-Anweisungen zu verwenden. Optional können Sie am Ende auch noch eine alternative `else`-Verzweigung hinzufügen. Hier ein Beispiel:

```
if(ausdruck1) {
   anweisung1;
}
```

```
else if (ausdruck2 ) {
   anweisung2;
}
else if (ausdruck3) {
   anweisung3;
}
//   Weitere else-if-Verzweigungen möglich
else {
   anweisung4;
}
...
```

Das folgende Listing zeigt eine solche `else-if`-Kette in der Praxis:

```
00  // kap005/listing004.c
01  #include <stdio.h>

02  int main(void) {
03    int ival = 0;
04    printf("Bitte eine positive Ganzzahl eingeben: ");
05    int check = scanf("%d", &ival);
06    if( check != 1 ) {
07      printf("Fehler bei der Eingabe!\n");
08    }
09    else if(ival <= 0) {
10      printf("Keine negativen Werte oder 0 verwenden\n");
11    }
12    else {
13      printf("Ihre Eingabe: %d\n", ival);
14    }
15    printf("Außerhalb der if-Verzweigung\n");
16    return 0;
17  }
```

Das Listing kennen Sie ja bereits in einer ähnlichen Form. Diesmal wurde allerdings gleich in der ersten bedingten `if`-Anweisung der Zeile **(06)** überprüft, ob auch ein gültiger Wert mit `scanf` passend zum Umwandlungszei-

chen eingegeben wurde. Wenn der Rückgabewert von `scanf` hier nicht 1 war, brauchen Sie gar nicht mehr weiterzumachen, weshalb hier gleich die Fehlermeldung in der Zeile **(07)** ausgegeben wird.

Wurde ein gültiger Wert eingegeben, prüfen Sie in der Zeile **(09)** mit der nächsten bedingten `else-if`-Anweisung den Inhalt von `ival`, ob der Wert kleiner oder gleich 0 war. Ist diese Bedingung wahr (ungleich 0), machen Sie nicht mehr weiter, weil Sie einen positiven Wert und auch keine 0 erhalten wollen. In diesem Fall geben Sie die Meldung in der Zeile **(10)** aus.

Wenn keine der mehrfach bedingten Verzweigungen zutrifft, kann nur noch die `else`-Verzweigung in den Zeilen **(12)** bis **(14)** ausgeführt werden. In dem Fall wird der gültige eingegebene Wert in der Zeile **(13)** ausgegeben.

Das Programm bei der Ausführung:

```
Bitte eine positive Ganzzahl eingeben: 123
Ihre Eingabe: 123
Außerhalb der if-Verzweigung

Bitte eine positive Ganzzahl eingeben: x
Fehler bei der Eingabe!
Außerhalb der if-Verzweigung

Bitte eine positive Ganzzahl eingeben: -77
Keine negativen Werte oder 0 verwenden
Außerhalb der if-Verzweigung
```

5.4.1 Verschachteln von Verzweigungen

Es ist natürlich auch möglich, solche Verzweigungen zu verschachteln. Je nach Situation können Sie hiermit das Programm verkürzen oder die Logik vereinfachen. Sie können aber leider auch das Gegenteil erreichen und unübersichtlichen Code erzeugen. Das folgende verschachtelte Beispiel, das dieselbe Funktionalität hat wie das Listing *listing004.c*, demonstriert dies:

```
00  // kap005/listing005.c
01  #include <stdio.h>
```

```
02  int main(void) {
03    int ival = 0;
04    printf("Bitte eine positive Ganzzahl eingeben: ");
05    int check = scanf("%d", &ival);
06    if( check == 1 ) {
07      if(ival <= 0) {
08        printf("Keine negativen Werte oder 0 verwenden\n");
09      }
10      else {
11        printf("Ihre Eingabe: %d\n", ival);
12      }
13    }
14    else {
15      printf("Fehler bei der Eingabe!\n");
16    }
17    printf("Außerhalb der if-Verzweigung\n");
18    return 0;
19  }
```

Zwar ist der logische Ablauf des Programms etwas anders als beim Listing *listing004.c* zuvor, aber das Endergebnis ist dasselbe. Zunächst wird in der Zeile **(06)** überprüft, ob der von `scanf` zurückgegeben Wert gleich 1 ist. Ist dies nicht der Fall, gibt der Ausdruck 0 zurück, und es kann gleich die alternative `else`-Anweisung in den Zeilen **(14)** bis **(16)** ausgeführt werden, weil ein Fehler bei der Eingabe aufgetreten ist. Wurde ein korrekter Wert mit `scanf` eingelesen, wird in der verschachtelten Verzweigung in **(07)** bis **(12)** der eingegebene Wert überprüft und ausgegeben, wenn kein negativer Wert oder 0 eingegeben wurde.

»if«-Verzweigungen ohne Anweisungsblock

Besitzt eine bedingte `if`-Anweisung, eine `else`-Verzweigung oder eine bedingte `else-if`-Verzweigung nur eine Anweisung, können Sie den Anweisungsblock mit `{ ... }` auch weglassen. Ein Anweisungsblock ist nur dann unbedingt nötig, wenn mehrere Anweisungen zu einem Block zusammengefasst werden müssen.

Sie sollten ein zu tiefes Verschachteln von Anweisungsblöcken nach Möglichkeit vermeiden. In der Regel können Sie Verschachtelungen umgehen, indem Sie das Design des Codes etwas überdenken. Im Zweifelsfall sollten Sie sich immer für den leichter lesbaren Code entscheiden.

Zwar gibt es keine Regeln, wie Sie Anweisungsblöcke anordnen, ich empfehle Ihnen aber, unbedingt eine saubere Formatierung wie beispielsweise gleichmäßige Einrückungen zu verwenden. Wenn Sie nicht umhinkommen, Anweisungsblöcke zu verschachteln, sind saubere Einrückungen ein Garant, dass Sie den Code auch noch in ein paar Wochen lesen können.

5.5 Mehrfache Verzweigung mit switch

In C finden Sie noch eine zweite Möglichkeit einer mehrfachen Verzweigung in Form der Fallunterscheidung `switch`. `switch` können Sie für die Auswertung eines ganzzahligen Ausdrucks verwenden. Die Ausdrücke zum Auswerten müssen `char`-, `int`- oder `long`-Werte sein. Die Syntax hierzu sieht folgendermaßen aus:

```
switch(Ausdruck)  {
  case Ausdruck1:  anweisungen1; break;
  case Ausdruck2:  anweisungen2; break;
  case Ausdruck3:  anweisungen3; break;
  …
  case AusdruckN:  anweisungenN; break;
         default:  anweisungen;
}
```

Mit `switch` wird hier der ganzzahlige `Ausdruck` bewertet und mit den ganzzahligen Konstanten der folgenden `case`-Marken verglichen. Stimmt eine dieser `case`-Marken mit der `switch`-Auswertung von `Ausdruck` überein, wird die Programmausführung hinter dieser `case`-Marke fortgeführt. Stimmt keine `case`-Marke mit der `switch`-Auswertung überein, kann optional eine `default`-Marke verwendet werden. Diese wird dann auf jeden Fall ausgeführt.

Hierzu ein vereinfachter Programmablaufplan einer solchen `switch`-Fallunterscheidung:

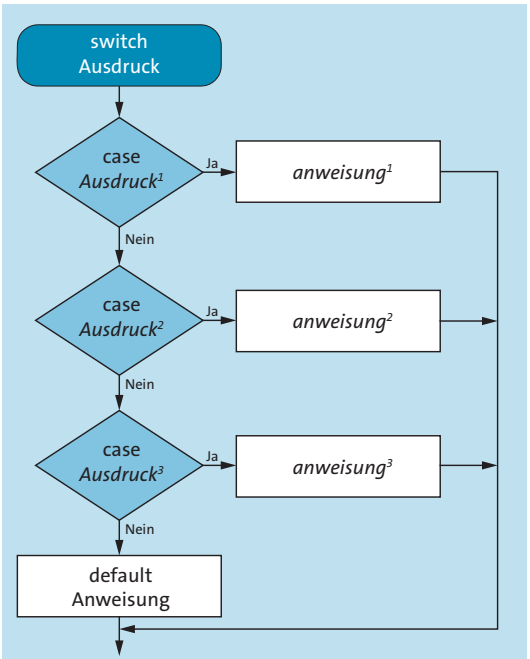

Abbildung 5.4 Ein einfacher Programmablaufplan einer »switch«-Fallunterscheidung

Folgendes Beispiel soll Ihnen die `switch`-Fallunterscheidung näherbringen:

```
00  // kap005/listing006.c
01  #include <stdio.h>

02  int main(void) {
03    int eingabe = 0;
04    printf("-1- Level 1\n");
05    printf("-2- Level 2\n");
```

```
06    printf("-3- Level 3\n");
07    printf("-4- Beenden\n");
08    printf("Ihre Auswahl bitte: ");
09    int check = scanf("%d", &eingabe);
10    if( check != 1) {
11      printf("Fehler bei der Eingabe ...\n");
12      return 1;
13    }
14    switch(eingabe) {
15      case 1  : printf("Level 1 war die Auswahl\n");
16                break;
17      case 2  : printf("Level 2 war die Auswahl\n");
18                break;
19      case 3  : printf("Level 3 war die Auswahl\n");
20                break;
21      case 4  : printf("Beenden war die Auswahl\n");
22                break;
23      default : printf("%d? Unbekanntes Level!\n", eingabe);
24    }
25    return 0;
26  }
```

In diesem Beispiel werden Sie in einer Art Menü aufgefordert, eine Zahl von 1 bis 4 einzugeben. In Zeile **(14)** wird dieser Ausdruck in der `switch`-Anweisung überprüft und im `switch`-Rumpf (Zeile **(15)** bis **(24)**) mit seinen `case`-Marken verglichen. Je nachdem, welche `case`-Marke zutrifft – hier 1, 2, 3 oder 4 –, werden die darauffolgenden Anweisungen ausgeführt. Haben Sie beispielsweise den ganzzahligen Wert 2 eingegeben, wird die Programmausführung mit der `case`-Marke 2 (Zeile **(17)**) fortgeführt. Das bedeutet im konkreten Fall, dass alle Anweisungen hinter dem Doppelpunkt dieser `case`-Marke bis zum nächsten `break` ausgeführt werden. Wurde ein anderer Wert als 1, 2, 3 oder 4 eingegeben, werden die Anweisungen der alternativen (aber optionalen) `default`-Anweisung (Zeile **(23)**) ausgeführt.

Das Programm bei der Ausführung:

```
-1- Level 1
-2- Level 2
-3- Level 3
-4- Beenden
Ihre Auswahl bitte: 3
Level 3 war die Auswahl

-1- Level 1
-2- Level 2
-3- Level 3
-4- Beenden
Ihre Auswahl bitte: 99
99? Unbekanntes Level!
```

Raus aus der Fallunterscheidung mit break

Von besonderer Wichtigkeit bei der `switch`-Fallunterscheidung sind die `break`-Anweisungen am Ende einer `case`-Marke. Mit diesem `break` weisen Sie das Programm an, aus dem `switch`-Rumpf herauszuspringen und mit der Programmausführung dahinter fortzufahren. Verwenden Sie nach einer `case`-Marke keine `break`-Anweisung, werden alle weiteren Anweisungen (auch die der `case`-Marken) im `switch`-Rumpf bis zum nächsten `break` oder bis zum Ende des Rumpfes ausgeführt.

Das bewirkt »break« in einer »switch«-Fallunterscheidung

Dank eines `break` an letzter Stelle einer `case`-Marke ist es möglich, ohne geschweifte Klammern in `case`-Blöcken auszukommen. Ohne einen `break` werden alle folgenden `case`-Anweisungen unabhängig von den Bedingungen ausgeführt.

Das absichtliche Weglassen von `break` kann allerdings durchaus gewollt sein, wie das folgende Beispiel demonstriert:

```
00  // kap005/listing007.c
01  #include <stdio.h>
```

```
02  int main(void) {
03    int opt = 0;
04    printf("-1- Option A\n");
05    printf("-2- Option B\n");
06    printf("-3- Option C\n");
07    printf("-4- Option D\n");
08    printf("Ihre Auswahl: ");
09    int check = scanf("%d", &opt);
10    if(check != 1) {
11      printf("Fehler bei der Eingabe ...\n");
12      return 1;
13    }
14    switch(opt) {
15      case 1  : printf("Option A beinhaltet auch ");
16      case 2  : printf("Option B\n");
17                break;
18      case 3  :
19      case 4  : printf("Option C und D sind gleich\n");
20                break;
21      default : printf("Unbekannte Option (%d)?\n" , opt);
22    }
23    return 0;
24  }
```

In Listing *listing007.c* wurde nach der Anweisung in Zeile **(15)** kein `break` verwendet. Falls nun der Wert 1 eingegeben wird, wird neben der `case`-Marke für 1 auch gleich die `case`-Marke für 2 aus Zeile **(16)** ausgeführt. Es wird hier einfach davon ausgegangen, dass für die Funktion mit der Option A zusätzlich noch die Funktion mit der Option B benötigt wird. Wird hingegen nur mit dem Wert 2 die Option B gewählt, dann werden nur die Anweisungen hinter der `case`-Marke von 2 (Zeilen **(16)** und **(17)**) ausgeführt. Zu den Funktionen der `case`-Marke 1 mit der Option A wird dabei nicht verzweigt.

In den Zeilen **(18)** und **(19)** wurde Ähnliches gemacht. Die leere `case`-Marke mit dem Wert 3 ohne `break` dient dazu, weil es hier egal ist, ob Sie den Wert

3 oder 4 eingeben; es werden immer dieselben Anweisungen der Zeilen **(19)** und **(20)** ausgeführt.

Das Programm bei der Ausführung:

```
-1- Option A
-2- Option B
-3- Option C
-4- Option D
Ihre Auswahl: 1
Option A beinhaltet auch Option B

-1- Option A
-2- Option B
-3- Option C
-4- Option D
Ihre Auswahl: 2
Option B

-1- Option A
-2- Option B
-3- Option C
-4- Option D
Ihre Auswahl: 3
Option C und D sind gleich

-1- Option A
-2- Option B
-3- Option C
-4- Option D
Ihre Auswahl: 4
Option C und D sind gleich
```

Natürlich sollte nicht unerwähnt bleiben, dass Sie anstatt einer `switch`-Fallunterscheidung auch bedingte `if`-Anweisungen mit `else-if`-Verzweigungen für unser Beispiel verwenden können. Wann Sie mehrfache Alternativen mit `switch` oder mit `else if` erstellen sollten und wann nicht, hängt natürlich auch vom Anwendungsfall ab.

5.6 Logische Verknüpfungen

Für komplexere Bedingungen (und später auch Schleifen) können sogenannte logische Operatoren verwendet werden. Damit können Sie mehrere Ausdrücke miteinander in einer Bedingung verknüpfen. Dies ist beispielsweise nötig, wenn ein Code nur dann ausgeführt werden soll, wenn zwei oder mehrere Bedingungen oder auch nur eine von mehreren Bedingungen zutreffen. Für solche Zwecke bietet C die logischen Operatoren UND, ODER und NICHT an. Die entsprechenden Symbole sind in der folgenden Tabelle 5.2 kurz beschrieben.

Operator	Bedeutung	Beispiel	Ergebnis
&&	UND-Operator	A **&&** B	Gibt ungleich 0 (wahr) zurück, wenn A **und** B ungleich 0 sind. Ansonsten wird 0 (unwahr) zurückgegeben.
||	ODER-Operator	A **||** B	Gibt ungleich 0 (wahr) zurück, wenn A **oder** B (oder beide) ungleich 0 sind. Ansonsten wird 0 (unwahr) zurückgegeben.
!	NICHT-Operator		Gibt ungleich 0 (wahr) zurück, wenn A **nicht** ungleich 0 ist. Ist A hingegen ungleich 0 (wahr), wird 0 zurückgegeben.

Tabelle 5.2 Logische boolesche Operatoren in C

5.6.1 Der !-Operator

Der logische NICHT-Operator (NOT) ist ein unärer Operator und wird gerne verwendet, um eine Bedingung auf einen Fehler hin zu testen.

Anstatt immer zu testen, ob eine bestimmte Bedingung gleich 0 zurückgibt, wird der !-Operator verwendet. Ein Beispiel:

```
if(ausdruck==0) { // Fehler }
```

Hier wird getestet, ob ausdruck gleich 0 ist. In der Praxis handelt es sich allerdings eher selten um eine Überprüfung, ob ein Ganzzahlwert einer Variablen gleich 0 ist, sondern ob ein bestimmter Ausdruck oder der Rückgabewert einer Funktion gleich 0 – also unwahr – und somit fehlerhaft ist. Daher finden Sie hier statt des Vergleichs mit dem ==-Operator auf 0 eher den logischen !-Operator. Nachfolgend die äquivalente Schreibweise mit dem logischen NICHT-Operator:

```
if ( !ausdruck ) { // Fehler }
```

Hierzu ein Beispiel, das eine einfache Passworteingabe in Form einer Geheimnummer überprüft:

```
00  // kap005/listing008.c
01  #include <stdio.h>

02  int main(void) {
03    int geheimnummer = 0;
04    printf("Geheimnummer eingeben: ");
05    int check = scanf("%d", &geheimnummer);
06    if( check != 1 ) {
07      printf("Fehler bei der Eingabe\n");
08      return 1;
09    }
10    else if( ! (geheimnummer == 123456) ) {
11      printf("Geheimnummer ist falsch!\n");
12    }
13    else {
14      printf("Geheimnummer ist richtig!\n");
15    }
16    return 0;
17  }
```

Der logische NICHT-Operator wird in Zeile **(10)** ausgeführt. Es wird überprüft, ob die Bedingung zwischen den Klammern, nämlich ob die Variable `geheimnummer` dem Wert 123456 entspricht, **nicht** zutrifft. Allerdings hätten Sie in diesem Fall keinen Vorteil, wenn Sie statt der Zeile **(10)** folgenden äquivalenten Code verwenden würden:

```
if(geheimnummer != 123456 )
```

In der Praxis ist es tatsächlich fast immer möglich, eine Alternative für den logischen NICHT-Operator zu verwenden. Häufiger als in unserem trivialen Beispiel in *listing008.c* wird der NICHT-Operator verwendet, um Funktionen auf eine erfolgreiche Ausführung hin zu überprüfen. Ein Pseudocode als Beispiel:

```
if(! funktion() ) {
  // Fehler bei der Funktionsausführung
}
```

Logischer NICHT-Operator im C99-Standard

Mit dem C99-Standard wurde als alternative Schreibweise für den logischen `!`-Operator das Makro `not` hinzugefügt, das in der Headerdatei `<iso646.h>` definiert ist. Somit würde die Schreibweise `if(!a)` exakt `if(not a)` entsprechen. Bezogen auf das Listing *listing008.c* würde die Zeile **(10)** mit dem Makro aber folgendermaßen aussehen:

```
#include <iso646.h>   // Benötigte Headerdatei für not
...
if( not (geheimnummer == 123456) )  {
  // Geheimnummer falsch
}
...
```

5.6.2 Der &&-Operator – Logisches UND

Mit dem logischen UND-Operator (`&&`) können Sie mehrere Operanden miteinander verknüpfen. Mehrere mit UND verknüpfte Anweisungen geben nur dann wahr – also ungleich 0 – zurück, wenn alle einzelnen Ope-

randen wahr sind. Ansonsten gibt der Ausdruck 0 – also unwahr – zurück. Hierzu ein Pseudocode:

```
if( (Bedingung1) && (Bedingung2) )  {
  // Beide Bedingungen sind wahr
}
else  {
  // Mindestens eine Bedingung ist 0 (also unwahr)
}
```

Das folgende Beispiel demonstriert den logischen UND-Operator in der Praxis:

```
00  // kap005/listing009.c
01  #include <stdio.h>

02  int main(void) {
03    int ival = 0;
04    printf("Eine Zahl von 1 bis 10 eingeben: ");
05    int check = scanf("%d", &ival);
06    if( check != 1 ) {
07      printf("Fehler bei der Eingabe\n");
08      return 1;
09    }
10    if( (ival > 0) && (ival <=10) )  {
11      printf("Zahl ist zwischen 1 und 10\n");
12    }
13    else  {
14      printf("Zahl ist nicht zwischen 1 und 10\n");
15    }
16    return 0;
17  }
```

In diesem Beispiel werden Sie aufgefordert, eine Zahl zwischen 1 und 10 einzugeben. In der Zeile **(10)** wird dann überprüft, ob der eingegebene Wert von `ival` größer als 0 **und** kleiner oder gleich 10 ist. Trifft beides zu, ist der Ausdruck wahr, und es wird Entsprechendes (Zeile **(11)**) ausgegeben. Andernfalls wird die Meldung aus Zeile (14) ausgegeben.

Das Programm bei der Ausführung:

```
Eine Zahl von 1 bis 10 eingeben: 3
Zahl ist zwischen 1 und 10

Ein Zahl von 1 bis 10 eingeben: 0
Zahl ist nicht zwischen 1 und 10

Ein Zahl von 1 bis 10 eingeben: 10
Zahl ist zwischen 1 und 10
```

Abbruch bei einer logischen UND-Verknüpfung

Ist bei einer logischen UND-Verknüpfung die linksstehende Bedingung gleich 0 (unwahr), wird im Gegensatz zum bitweisen UND (&) die rechtsstehende Bedingung nicht mehr überprüft, weil der Gesamtausdruck unwahr ist.

5.6.3 Der ||-Operator – Logisches ODER

Benötigen Sie hingegen eine logische Verknüpfung, bei der der gesamte Ausdruck wahr zurückgibt, wenn nur mindestens einer der verknüpften Operanden wahr ist, dann können Sie dies mit dem ODER-Operator (||) realisieren. Auch hierzu ein kurzer Pseudocode, damit Sie den logischen ODER-Operator besser verstehen:

```
if( (Bedingung1) || (Bedingung2) )  {
  // Mindestens Bedingung1 oder Bedingung2 ist wahr.
}
else  {
 // Beide Bedingungen sind unwahr.
}
```

Natürlich gibt es auch hierzu wieder ein Codebeispiel, um den ODER-Operator in der Praxis kennenzulernen:

```
00  // kap005/listing010.c
01  #include <stdio.h>
```

```
02  int main(void) {
03    unsigned int uval1 = 0, uval2 = 0;
04    printf("Bitte zwei Ganzzahlen eingeben: ");
05    int check = scanf("%u %u", &uval1, &uval2);
06    if(check != 2) {
07      printf("Fehler bei der Eingabe...\n");
08      return 1;
09    }
10    if( (!uval1) || (!uval2) ) {
11      printf("Fehler: Einer der Werte ist gleich 0\n");
12    }
13    else  {
14      printf("%u / %u = %lf\n",
            uval1, uval2, (double)uval1/uval2);
15    }
16    return 0;
17  }
```

In diesem Beispiel sollen zwei Ganzzahlen `uval1` und `uval2` dividiert werden. Dafür werden die beiden eingegebenen Ganzzahlen in der Zeile (**10**) mit dem logischen NICHT-Operator überprüft. Ist mindestens eine der beiden Zahlen gleich Null, wird die Fehlermeldung in der Zeile (**11**) ausgegeben und die Berechnung in der Zeile (**14**) nicht ausgeführt.

Abbruch bei einer logischen ODER-Verknüpfung

Ist bei einer logischen Verknüpfung eine Bedingung gleich 1 (wahr), werden weitere damit verknüpfte Bedingungen nicht mehr überprüft, weil der Gesamtausdruck wahr ist.

Das Programm bei der Ausführung:

```
Bitte zwei Ganzzahlen eingeben: 10 0
Fehler: Einer der Werte ist gleich 0

Bitte zwei Ganzzahlen eingeben: 10 4
10 / 4 = 2.500000
```

&& und || miteinander mischen und verknüpfen

Sie können natürlich auch mit dem `&&`-Operator und dem `||`-Operator weitere Bedingungen miteinander verknüpfen. Hierbei sollten Sie aber stets die Lesbarkeit des Codes im Auge behalten.

Logische UND- und ODER-Makros im C99-Standard

Im C99-Standard wurden für die logischen Operator-Gegenstücke `&&` und `||` die Makros `and` und `or` eingeführt. Sie sind in der Headerdatei `<iso646.h>` definiert. Bezogen auf das Listing *listing010.c* würde somit die Zeile **(10)** mit den Makros wie folgt aussehen:

```
#include <iso646.h> // Benötigte Headerdatei
...
if( (not uval1) or (not uval2) )
```

5.7 Kontrollfragen und Aufgaben

1. Welchen Wert gibt eine bedingte `if`-Anweisung zwischen den Klammern `()` zurück, wenn der Ausdruck der Bedingung richtig oder falsch ist?
2. Wie lautet die alternative Verzweigung einer `if`-Anweisung, und wann wird diese ausgeführt?
3. Was können Sie verwenden, wenn Sie mehrere Verzweigungen benötigen?
4. Welche besondere Bedeutung spielt die Anweisung `break` in einem `switch`-Konstrukt?
5. Wird keine passende `case`-Marke in einem `switch`-Konstrukt angesprungen, findet keine Verarbeitung statt. Wie können Sie trotzdem eine optionale Marke im `switch`-Konstrukt einbauen, die ausgeführt wird, wenn in keine `case`-Marke gesprungen wird?
6. In C gibt es drei logische Operatoren. Nennen Sie diese, und geben Sie an, wozu sie in der Regel verwendet werden.

7. Finden Sie heraus, ob die logischen Verknüpfungen 1 (wahr) oder 0 (unwahr) ergeben. Versuchen Sie, die Verknüpfungen ohne ein Programm für die Ausgabe der einzelnen Werte zu lösen.

```
int  ival1 = 11, ival2 = 22;

int logo1 = (ival1 == 11) && (ival2 != 11);
int logo2 = (ival1 != 11) || (ival2 != 11);
int logo3 = (!(ival1 != ival2)) && (!(ival2-ival2));
int logo4 = (ival1 < ival2) && (ival2 != 22);
int logo5 = (!(ival1 == ival2)) || (!(ival1 < ival2));
```

8. Erstellen Sie ein Listing, das überprüft, ob eine gerade oder ungerade Zahl eingegeben wurde (**Tipp:** %-Operator verwenden). Beschränken Sie außerdem die Zahlen auf einen Bereich von 1 bis 100 (**Tipp:** logische Operatoren verwenden).

9. Schreiben Sie das folgende Programm um, damit eine `switch`-Fallunterscheidung statt der vielen `if`-Anweisungen verwendet wird.

```
// kap005/aufgabe001.c
#include <stdio.h>

int main(void) {
  int work = 0;
  printf("-1- PC 1 hochfahren\n");
  printf("-2- PC 2 hochfahren\n");
  printf("-3- Drucker einschalten\n");
  printf("-4- Kaffee machen\n");
  printf("-5- Feierabend machen\n");
  printf("Was wollen Sie tun: ");
  if( scanf("%d", &work) != 1 ) {
    printf("Fehler bei der Eingabe...\n");
    return 1;
  }
  if( work == 1 ) {
    printf("PC 1 wird hochgefahren\n");
  }
  else if( work == 2 ) {
```

```
18      printf("PC 2 wird hochgefahren\n");
19    }
20    else if( work == 3 ) {
21      printf("Drucker wird eingeschaltet\n");
22    }
23    else if( work == 4 ) {
24      printf("Kaffee wird gemacht\n");
25    }
26    else if( work == 5 ) {
27      printf("Gute Nacht\n");
28    }
29    else {
30      printf("Falsche Eingabe!\n");
31    }
32    return 0;
33  }
```

Kapitel 6

Schleifen – Programmteile wiederholen

Wenn Sie eine Gruppe von Anweisungen mehrfach ausführen wollen, stellt Ihnen C mit `for`, `while` und `do while` drei verschiedene Schleifen – sogenannte Iterationsanweisungen oder auch Wiederholungen – zur Verfügung.

6.1 Die Zählschleife – for

Die `for`-Schleife wird häufig als Zählschleife bezeichnet. Zunächst die Syntax dieser Schleife:

```
for( Initialisierung; Bedingung; Reinitialisierung ) {
   // Anweisung(en)
}
```

Beim Eintritt in die `for`-Schleife wird vor dem eigentlichen Schleifendurchlauf einmalig die `Initialisierung` ausgeführt. Gewöhnlich wird hierbei die Schleifenvariable initialisiert. Es kann stattdessen aber auch eine beliebige Anweisung ausgeführt werden. `Bedingung` ist die logische Bedingung, welche den Schleifenablauf regelt, also die Abbruchbedingung für die Schleife festlegt. Solange `Bedingung` wahr (ungleich 0) ist, werden die Anweisungen im Schleifenrumpf ausgeführt. Ist `Bedingung` hingegen unwahr (gleich 0), endet die Schleife, und die Programmausführung wird hinter dem Schleifenrumpf fortgeführt. Der letzte Ausdruck, die `Reinitialisierung`, wird immer zum Abschluss eines jeden Schleifendurchgangs ausgeführt. In der Regel wird die Reinitialisierung für die Veränderung der Schleifenvariable genutzt. Allerdings kann hierfür auch eine beliebige Anweisung verwendet werden.

Zum besseren Verständnis zeigt Abbildung 6.1 einen logischen Programmablaufplan der `for`-Schleife.

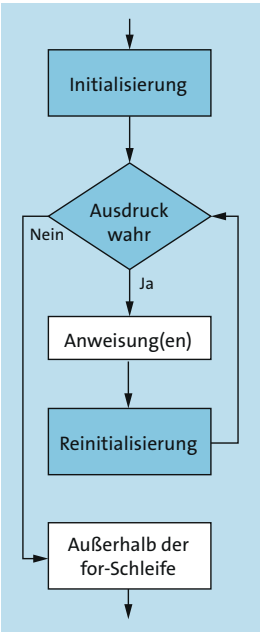

Abbildung 6.1 Programmablaufplan für die übliche Verwendung einer »for«-Schleife

Dieses Minimalbeispiel demonstriert die `for`-Schleife bei der Ausführung:

```
00  // kap006/listing001.c
01  #include <stdio.h>

02  int main(void) {
03    for( int cnt = 1; cnt <= 5; cnt++ ) {
04      printf("%d. Schleifendurchlauf\n", cnt);
05    }
06    return 0;
07  }
```

Der Schleifenablauf spielt sich in den Zeilen **(03)** bis **(05)** ab. In der Schleife wird zunächst die Variable `cnt` mit dem Wert 1 initialisiert. Anschließend

wird überprüft, ob der Wert der Variablen `cnt` kleiner oder gleich 5 ist. Ist dies der Fall, wird die Anweisung im Schleifenrumpf (Zeile **(04)**) ausgeführt. Nach der Ausführung der Anweisung wird der dritte Ausdruck der `for`-Schleife ausgeführt. Hier wird der Wert der Variablen mit `cnt++` um den Wert 1 erhöht. Jetzt wird wieder überprüft, ob der Wert von `cnt` kleiner oder gleich 5 ist.

Der Vorgang wird so lange wiederholt, bis die Abbruchbedingung, dass `cnt` kleiner oder gleich 5 ist, unwahr (also gleich 0) zurückliefert. Ist dies der Fall, wird der Schleifendurchlauf beendet, und es wird mit der Programmausführung hinter der Schleifenanweisung (hier Zeile **(06)**) fortgefahren.

Das Programm bei der Ausführung:

```
1. Schleifendurchlauf
2. Schleifendurchlauf
3. Schleifendurchlauf
4. Schleifendurchlauf
5. Schleifendurchlauf
```

Das Beispiel soll nicht darüber hinwegtäuschen, dass die `for`-Schleife extrem vielseitig und flexibel ist. So können Sie den Schleifendurchlauf von *listing001.c* auch wie folgt ändern:

```
for(int cnt=1;cnt<=5;printf("%d. Schleifendurchlauf\n",cnt++));
```

Hier wurde die Reinitialisierung der Schleifenvariablen gleichzeitig mit der `printf`-Anweisung im dritten Argument der `for`-Schleife verwendet. So können Sie sich den Schleifenrumpf hier auch gleich sparen und mit einem Semikolon abschließen. Allerdings sollte dieses Beispiel keine Schule machen. Es dient nur zu Demonstrationszwecken.

Neben der Möglichkeit, verschiedene Ausdrücke in der `for`-Schleife zu verwenden, können auch Ausdrücke fehlen. Entscheidend ist, dass die beiden Semikola in den Klammern an der richtigen Stelle vorhanden sind. Alle folgenden Beispiele sind erlaubt:

```
// ohne 1. Ausdruck
for( ; i < 10; i++ ) { ... }
```

```
// ohne 1. und 3. Ausdruck
for( ; i < 10; ) { ... }
// ohne einen Ausdruck - eine Endlosschleife
for( ;; ) { ... }
```

Beachten Sie allerdings, dass dann die Schleifenvariable, die Sie innerhalb von `for` definiert haben, nicht mehr nach dem Ende des Anweisungsblockes der `for`-Schleife zur Verfügung steht. Sie haben allerdings ggf. auch den Vorteil, ein versehentliches Semikolon am Ende einer `for`-Schleife zu entdecken, beispielsweise:

```
for(int i=0; i<10; i++);   // <- fehlerhaftes Semikolon
   printf("%d\n", i);  // wird erst nach Schleife ausgeführt
```

Hier war wohl eher geplant, `printf` bei jedem Schleifendurchlauf aufzurufen, um den aktuellen Wert von `i` auszugeben. Stattdessen wird aufgrund des (hier) irrtümlich gesetzten Semikolons am Ende von `for` die Schleife zwar ausgeführt, aber erst danach `printf` nur ein einziges Mal mit dem aktuellen Wert von `i` aufgerufen und ausgegeben. Das Beispiel enthält keinen syntaktischen, sondern einen logischen Fehler, der wesentlich schwieriger aufzufinden ist, weil für den Compiler alles in Ordnung ist.

Definieren Sie hingegen die Schleifenvariable `i` innerhalb von `for`, würde der Compiler sich mit einer Fehlermeldung beschweren, wenn Sie irrtümlicherweise ein Semikolon am Ende von `for` verwendet hätten, weil die Variable nicht mehr im Gültigkeitsbereich läge:

```
for(int i=0; i<10; i++);   // <- fehlerhaftes Semikolon
   printf("%d\n", i);      // Fehler: i ist hier unbekannt
```

Es ist auch möglich, im Schleifenkopf von `for` mehrere Ausdrücke mit dem **Kommaoperator** getrennt zu verwenden. In der Praxis werden so zum Beispiel in einer Anweisung (vor dem ersten Semikolon) mehrere Variablen initialisiert und/oder in der letzten Anweisung mehrere Variablen reinitialisiert. Hierzu ein einfaches Beispiel:

```
00  // kap006/listing002.c
01  #include <stdio.h>
```

```
02  int main(void) {
03    for( int i=1, j=10; i < j; i++, j--) {
04      printf("i=%d, j=%d\n", i, j);
05    }
06    return 0;
07  }
```

Im Schleifenkopf der Zeile **(03)** werden die beiden `int`-Variablen `i` und `j` zunächst mit einer Initialisierungsanweisung mit Werten initialisiert. Im Reinitialisierungsausdruck werden die Werte dieser Variablen dann inkrementiert bzw. dekrementiert. Im Beispiel wird einfach der eine Wert hoch- und der andere heruntergezählt, bis `i` nicht mehr kleiner als `j` ist (was die Abbruchbedingung der Schleife ist). In der Praxis findet man solche Schleifenkonstrukte in Such- oder Sortieralgorithmen.

Das Programm bei der Ausführung:

```
i=1, j=10
i=2, j=9
i=3, j=8
i=4, j=7
i=5, j=6
```

Erwähnt werden sollte noch, dass Sie `for`-Schleifen natürlich auch ineinander verschachteln können. Solche verschachtelten Schleifen finden Sie ebenfalls häufiger in Such- oder Sortieralgorithmen. Hier ein Beispiel:

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

6.2 Die kopfgesteuerte while-Schleife

Die `while`-Schleife ist eine kopfgesteuerte Schleife und führt einen Block von Anweisungen so lange aus, wie die Schleifenbedingung wahr ist. Die Syntax der `while`-Schleife sieht folgendermaßen aus:

```
while( Ausdruck ) {
   Anweisung(en)
}
```

Bevor ein Schleifendurchlauf gestartet wird, wird zunächst `Ausdruck` ausgewertet. Solange dieser wahr ist (ungleich 0), werden die Anweisungen im Schleifenrumpf ausgeführt. Ist die Schleifenbedingung unwahr (gleich 0), wird die Schleife beendet, und das Programm fährt mit der Ausführung dahinter fort. Der typische Programmablaufplan für die `while`-Schleife sieht demnach wie folgt aus:

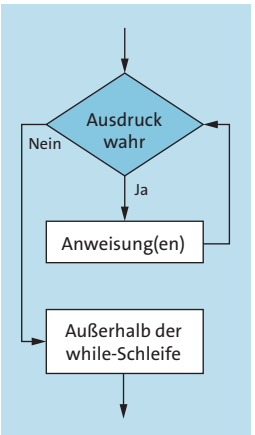

Abbildung 6.2 Programmablaufplan der »while«-Schleife

Ein einfaches Beispiel:

```
00  // kap006/listing003.c
01  #include <stdio.h>

02  int main(void) {
03    int ival = 10;
04    while( ival > 0 ) {
05      if( ival % 2 ) {
06        printf("%d ", ival);
07      }
```

```
08      ival--;
09    }
10    printf("\n");
11    return 0;
12  }
```

In Zeile **(03)** wird die Schleifenvariable `ival` mit dem Wert 10 initialisiert. Eine Zeile **(04)** weiter geht es in die `while`-Schleife. Es wird die Bedingung überprüft, ob der Wert von `ival` größer als 0 ist. Solange diese Bedingung wahr ist (ungleich 0), wird der Schleifenrumpf von der Zeile **(05)** bis **(09)** ausgeführt. In der Schleife selbst wird in der Zeile **(05)** lediglich mit dem Modulo-Operator überprüft, ob der aktuelle Wert von `ival` durch 2 geteilt einen Rest ergibt. Ist dies der Fall, handelt es sich um eine ungerade Zahl, und die geben Sie in Zeile **(06)** aus. Ansonsten wird nichts ausgegeben.

»while«-Schleife zu »for«-Schleife

Aus jeder `while`-Schleife können Sie natürlich auch eine `for`-Schleife formen. Umgekehrt funktioniert dies genauso.

In jedem Schleifendurchlauf wird – und das ist in diesem Beispiel besonders wichtig – der Wert von `ival` in Zeile **(08)** um 1 reduziert. Ohne diese Dekrementierung würde sich die Schleife nie mehr ohne fremde Hilfe beenden lassen – die Abbruchbedingung würde nie erreicht. Nachdem der Wert von `ival` um 1 reduziert wurde, geht es wieder hoch zum Schleifenanfang in der Zeile **(04)**, wo die Bedingung erneut überprüft wird. Der Vorgang wird so lange ausgeführt, bis die Bedingung in der Zeile **(04)** unwahr ist (gleich 0). Trifft dies zu, wird mit der Ausführung des Programms hinter dem Rumpf der `while`-Schleife mit der Zeile **(10)** fortgefahren.

Das Programm bei der Ausführung:

```
9 7 5 3 1
```

6.3 Die fußgesteuerte do-while-Schleife

Das Gegenstück zur kopfgesteuerten `while`-Schleife ist die fußgesteuerte `do-while`-Schleife. Hier die Syntax dieser dritten und letzten Schleife:

```
do {
   Anweisung(en)
} while( Ausdruck );
```

Beim Ausführen der `do-while`-Schleife werden zunächst die Anweisungen im Schleifenrumpf abgearbeitet. Somit garantiert Ihnen diese Schleife, dass der Schleifenrumpf bzw. der Anweisungsblock mindestens einmal ausgeführt werden. Anschließend wird mit `while` der Ausdruck ausgewertet. Ist dieser wahr (ungleich 0), werden erneut die Anweisungen hinter dem Schlüsselwort `do` ausgeführt. Gibt die Schleifenbedingung hingegen unwahr (gleich 0) zurück, wird die Schleife beendet, und das Programm fährt mit der Ausführung dahinter fort.

Anwendungszweck für »do while«

Die `do-while`-Schleife wird hauptsächlich dann verwendet, wenn Anweisungen in einer Schleife mindestens einmal, aber eventuell auch mehrmals ausgeführt werden müssen. Da die `do-while`-Schleife die Schleifenbedingung erst am Ende abfragt, ist dies – anders als bei `for`/`while`-Schleifen – garantiert.

Den Programmablaufplan der `do-while`-Schleife sehen Sie in Abbildung 6.3.

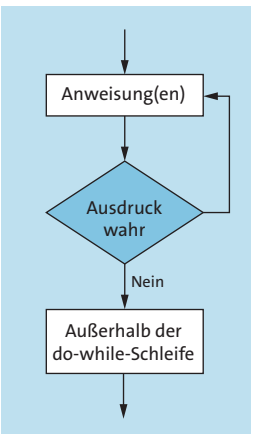

Abbildung 6.3 Programmablaufplan der »do while«-Schleife

Hier ein einfaches Listing zur do-while-Schleife:

```
00  // kap006/listing004.c
01  #include <stdio.h>

02  int main(void) {
03    int auswahl = 0;
04    do {
05      printf("-1- Funktion 1 verwenden\n");
06      printf("-2- Funktion 2 verwenden\n");
07      printf("-3- Funktion 3 verwenden\n");
08      printf("-0- Programm beenden\n\n");
09      printf("Ihre Auswahl :\t");
10      if( scanf("%d", &auswahl) != 1) {
11        printf("Fehler bei der Eingabe...\n");
12        return 1;
13      }
14      switch( auswahl ) {
15        case 1 : printf("Funktion 1 bei der Arbeit\n");
16                 break;
17        case 2 : printf("Funktion 2 bei der Arbeit\n");
18                 break;
19        case 3 : printf("Funktion 3 bei der Arbeit\n");
20                 break;
21        case 0 : printf("Beende Programm\n");
22                 break;
23        default: printf("Falsche Eingabe!!!\n");
24      }
25    } while( auswahl != 0 );
26    printf("Auf Wiedersehen!\n");
27    return 0;
28  }
```

Die do-while-Schleife wird mit dem Schlüsselwort do in der Zeile **(04)** eingeleitet. Anschließend wird der komplette Code bis zur Zeile **(25)** wie gewöhnlich ausgeführt. Im Beispiel wurde eine switch-Fallunterscheidung verwendet. Das ist eine Art Menüführung für die Konsole, in der Sie eine

bestimmte Funktion über die Eingabe einer numerischen Ganzzahl auswählen können. In Zeile **(25)** wird diese numerische Ganzzahl `auswahl` als Schleifenabbruchbedingung ausgewertet. Solange der Anwender hier nicht die numerische Ganzzahl 0 eingegeben hat, wird die `do-while`-Schleife erneut ab der Zeile **(04)** und damit das Programm weiter ausgeführt. Mit der Eingabe der Ganzzahl 0 können Sie das Programm ordentlich beenden und mit der Zeile **(26)** im Programm fortfahren.

Das Programm bei der Ausführung:

```
-1- Funktion 1 verwenden
-2- Funktion 2 verwenden
-3- Funktion 3 verwenden
-0- Programm beenden

Ihre Auswahl :       2
Funktion 2 bei der Arbeit
-1- Funktion 1 verwenden
-2- Funktion 2 verwenden
-3- Funktion 3 verwenden
-0- Programm beenden

Ihre Auswahl :      0
Beende Programm
Auf Wiedersehen!
```

6.4 Kontrollierte Sprünge aus Schleifen

Wenn Sie eine Schleife beenden oder wieder an den Anfang des Schleifenkopfes springen möchten, können Sie die Schlüsselwörter `break` oder `continue` verwenden.

break

Das Schlüsselwort `break` haben Sie bereits bei der `switch`-Fallunterscheidung kennengelernt. `break` kann auch verwendet werden, um eine `for`-, `while` oder `do-while`-Schleife vorzeitig zu verlassen. Beachten Sie dabei, dass bei einer verschachtelten Schleife nur die innerste Schleife abgebrochen wird.

Hier ein einfaches Beispiel zum Schlüsselwort `break`:

```
00 // kap006/listing005.c
01 #include <stdio.h>

02 int main(void) {
03   int ival = 0;
04   while( 1 ) {
05     printf("Raus geht es mit 5: ");
06     if( scanf("%d", &ival ) != 1 ) {
07       printf("Fehler bei der Eingabe...\n");
08       break;
09     }
10     if( ival == 5 ) {
11       break;
12     }
13   }
14   printf("Programmende\n");
15   return 0;
16 }
```

Aus dieser (Endlos-)Schleife aus Zeile **(04)** bis **(13)** geht es nur dann heraus, wenn Sie den numerischen Wert 5 eingeben und die `if`-Bedingung in der Zeile **(10)** wahr ist. Erst dann wird die `break`-Anweisung in der Zeile **(11)** ausgeführt. Dann wird die `while`-Schleife verlassen und mit der Programmausführung in der Zeile **(14)** fortgefahren. Alternativ wird hier die Schleife in der Zeile **(08)** ebenfalls mit `break` abgebrochen, wenn bei der Eingabe von `scanf` in der Zeile **(06)** kein gültiger Wert eingegeben wurde und der Rückgabewert nicht 1 war.

Allerdings besteht bei einem solchen Beispiel keine Notwendigkeit, eine `break`-Anweisung zu verwenden. Die Schleife aus dem Listing *listing005.c* mit den Zeilen **(04)** bis **(13)** hätte auch wie folgt erstellt werden können:

```
int ival = 0;
while( ival != 5 ) {
   printf("Raus geht es mit 5: ");
   if( scanf("%d", &ival ) != 1 ) {
```

```
      printf("Fehler bei der Eingabe...\n");
      break;
    }
  }
```

continue

Mit dem Schlüsselwort `continue` können Sie im Gegensatz zum Schlüsselwort `break` den Rest des aktuellen Schleifendurchlaufs überspringen und mit dem nächsten Schleifendurchlauf fortfahren. Hierzu ein Beispiel:

```
00  // kap006/listing006.c
01  #include <stdio.h>

02  int main(void) {
03    int sum=0;
04    for(int ival=0; ival < 20; ival++) {
05      if(ival % 2) {
06        continue;
07      }
08      sum+=ival;
09    }
10    printf("Summe gerader Zahlen: %d\n", sum);
11    return 0;
12  }
```

In diesem Beispiel wird die Summe aller geraden Zahlen addiert. Ist der Wert der Schleifenvariablen in Zeile **(05)** eine ungerade Zahl, wird ein Rest zurückgegeben, sodass diese Bedingung wahr (ungleich 0) ist. Dann wird mithilfe von `continue` in Zeile **(06)** wieder zum nächsten Schleifendurchlauf von `for` hochgesprungen, und die Zeile **(08)** dahinter – das Addieren von geraden Ganzzahlen – wird nicht ausgeführt.

Auch in diesem Beispiel könnten Sie auf die `continue`-Anweisung verzichten. Dieselbe Schleife, nur jetzt ohne die `continue`-Anweisung, könnten Sie wie folgt realisieren:

```
for(int ival=0; ival < 20; ival++) {
  if(! (ival % 2) ) {
```

```
      sum+=ival;
   }
}
```

Codeoptimierung: Wenn Sie sich den logischen Sachverhalt des Listings noch etwas genauer ansehen, könnten Sie auch auf die `if`-Verzweigung verzichten und gleich dafür sorgen, dass beim Hochzählen in der Schleife immer nur gerade Zahlen verwendet werden. Somit könnten Sie mit folgendem Schleifenkonstrukt auf alle `if`-Überprüfungen und die Hälfte der Schleifendurchläufe verzichten:

```
for(int ival=0; ival < 20; ival+=2) {
   sum+=ival;
}
```

Schleifen ohne Anweisungsblock

Besitzt der Schleifenrumpf nur eine Anweisung, können Sie den Anweisungsblock mit `{ ... }` auch weglassen. Ein Anweisungsblock ist nur dann nötig, wenn mehrere Anweisungen zu einem Block zusammengefasst werden müssen.

6.5 Kontrollfragen und Aufgaben

1. Was sind Schleifen?
2. Welche Schleifen stehen Ihnen zur Verfügung?
3. Welche Schleife sollten Sie verwenden, wenn mehrere Anweisungen mindestens einmal ausgeführt werden sollen?
4. Wie können Sie den normalen Schleifenablauf beeinflussen?
5. Was gibt diese Schleife aus, und welcher Fehler wurde hier gemacht?

```
int ival = 0;
while ( ival > 10 ) {
  printf("%d\n", ival);
  ival++;
}
```

6. Auf den ersten Blick scheint bei dieser Schleife alles in Ordnung zu sein. Auch logisch liegt hier kein Fehler vor. Warum läuft diese Schleife trotzdem in einer Endlosschleife, und was können Sie dagegen tun?

```
for(float fval = 0.0f; fval != 1.0f; fval+=0.1f) {
   printf("%f\n", fval);
}
```

7. Im folgenden Beispiel wird nur einmal 0 ausgegeben, und dann hängt das Programm in einer Endlosschleife fest. Was wurde falsch gemacht?

```
int ival = 0;
while ( ival < 20 ) {
  if(ival % 2) {
     continue;
  }
  printf("%d\n", ival);
  ival++;
}
```

8. Ein Kunde legt einen bestimmten Geldbetrag auf einem Konto an und bekommt dafür pro Jahr einen bestimmten Zinsanteil. Erstellen Sie ein Programm, das abfragt, wie viel Geld der Kunde auf das Konto einzahlt und welchen Zinssatz er dafür bekommt. Listen Sie mithilfe einer Schleife auf, wie sich das Geld Jahr für Jahr vermehrt. Natürlich fragen Sie den Anwender auch, wie viele Jahre er auflisten lassen will.

Kapitel 7

Funktionen erstellen

In der Praxis werden Sie die komplette Programmausführung nicht in einem einzigen Anweisungsblock zusammenfassen, sondern eine Problemlösung in viele kleine Teilprobleme zerlegen und mithilfe von mehreren Unterprogrammen, auch Funktionen genannt, lösen. Anweisungen eines Programms werden grundsätzlich in Funktionen zusammengefasst. Die Funktion, die beim Start eines Programmes (Startup) immer als erstes angesprochen wird, ist die `main`-Funktion. Diese haben Sie bisher immer verwendet. Die `main`-Funktion bildet die Hauptfunktion, und von dort aus werden alle anderen Funktionen aufgerufen und gestartet.

7.1 Funktionen definieren

Zunächst die grundlegende Syntax einer Funktion:

```
[Spezifizierer] <Datentyp> <Funktionsname>( [Parameter] ) {
   // Anweisung(en)
}
```

Im Funktionskopf können folgende Bestandteile verwendet werden:

- Spezifizierer: Ein Speicherklassen-Spezifizierer ist optional und wird gesondert in den entsprechenden Stellen des Buches beschrieben (beispielsweise in Abschnitt 7.6, »Exkurs: Funktion bei der Ausführung«)
- Datentyp: Hiermit wird der Typ des Rückgabewertes spezifiziert. Sie können beliebige Datentypen verwenden oder `void`, wenn eine Funktion keinen Rückgabewert zurückgibt.
- Funktionsname: Dies muss ein eindeutiger Funktionsname sein, mit dem Sie diese Funktion von einer anderen Programmstelle aus aufrufen können. Für den Namen gelten dieselben Regeln wie schon bei den Variablennamen (siehe Abschnitt 2.4.1, »Bezeichner«). Außerdem soll-

ten Sie keine Funktionsnamen der Laufzeitbibliothek, wie zum Beispiel `printf()` verwenden.

- Parameter: Auch die Parameter einer Funktion sind optional. Die Klammerung hinter dem Funktionsnamen allerdings nicht. Sie muss immer vorhanden sein. Wenn Sie keine Parameter verwenden, sollten Sie das Schlüsselwort `void` einsetzen. Ansonsten werden die einzelnen Parameter mit ihrem Datentyp und dem Bezeichner für die Verwendung innerhalb der Funktion spezifiziert und, wenn Sie mehr als einen Parameter verwenden, mit einem Komma voneinander getrennt.
- Anweisungsblock mit Anweisungen: Ebenso wie die `main`-Funktion enthält auch eine Funktion einen Anweisungsblock, in dem die Funktionsanweisungen und lokalen Deklarationen zusammengefasst werden.

7.2 Funktionen aufrufen

Nachdem Sie die grundlegenden Elemente einer Funktion kennengelernt haben, soll im folgenden Beispiel eine solche erstellt und aufgerufen werden:

```
00  // kap007/listing001.c
01  #include <stdio.h>

02  void hallo(void) {
03    printf("In der Funktion\n");
04  }

05  int main(void) {
06    printf("Vor der Funktion\n");
07    hallo();  // Funktionsaufruf
08    printf("Nach der Funktion\n");
09    return 0;
10  }
```

Wenn Sie das Programm starten, wird zunächst die `main`-Funktion ausgeführt. In Zeile **(06)** erfolgt eine `printf`-Ausgabe. Anschließend wird in Zeile **(07)** die Funktion mit dem entsprechenden Funktionsnamen und den

runden Klammern aufgerufen. In C erkennen Sie einen Funktionsaufruf daran, dass hinter dem Funktionsnamen die runden Klammern folgen (beispielsweise `funcname()`).

Nach dem Aufruf der Funktion wird die Funktion mit den Zeilen **(02)** bis **(04)** ausgeführt. Im Beispiel wird wiederum nur eine `printf`-Anweisung auf dem Bildschirm ausgegeben. Wenn die Funktion mit der Ausführung fertig ist, wird hinter dem Funktionsaufruf in der Zeile **(08)** der `main`-Funktion mit der Programmausführung fortgefahren.

Das Programm bei der Ausführung:

```
Vor der Funktion
In der Funktion
Nach der Funktion
```

7.3 Funktionsdeklaration (Vorausdeklaration)

Sie können Funktionen auch hinter den aufrufenden Funktionen definieren – also erst die `main`-Funktion erstellen und dahinter die Funktionsdefinition. Da allerdings ein Programm von oben nach unten durchgearbeitet wird, kann das Programm zum Zeitpunkt des Aufrufes diese Funktion nicht kennen. Hier ein Beispiel wie es nicht geht:

```
00  // kap007/listing002.c
01  #include <stdio.h>

02  int main(void) {
03    printf("Vor der Funktion\n");
04    hallo();  // Fehler/Warnung: hallo() unbekannt
05    printf("Nach der Funktion\n");
06    return 0;
07  }

08  void hallo(void) {
09    printf("In der Funktion\n");
10  }
```

Wenn das Programm in Zeile **(04)** die Funktion `hallo()` aufruft, ist diese dem Programm eigentlich noch gar nicht bekannt, weil sie erst ab Zeile **(08)** definiert wird. Gewöhnlich sollten Sie schon während der Übersetzung eine Warn- bzw. Fehlermeldung erhalten haben. Wenn sich das Listing in diesem Beispiel trotzdem ausführen lässt, liegt daran, dass der Compiler hier eine implizite Umwandlung des Rückgabewertes nach `int` durchführt, wenn eine Funktion vorher nicht deklariert oder definiert wurde. Sobald Sie eine Funktion mit einem anderen Rückgabewert als `void` oder `int` verwendet haben, klappt es auch nicht mehr mit der impliziten Konvertierung des Rückgabewertes.

Um ein fehlerhaftes Verhalten des Programms von vornherein zu vermeiden, haben Sie zwei Möglichkeiten:

- Sie **definieren** die Funktion **vor dem ersten Aufruf**, wie Sie dies zum Beispiel im Listing *listing001.c* in diesem Kapitel gemacht haben.
- Sie **deklarieren** die Funktion bzw. genauer den *Funktionsprototyp* **vor dem ersten Aufruf**. Bei einer Deklaration müssen Sie die Funktion nicht vollständig definieren, sondern nur den Funktionskopf angeben und mit einem Semikolon abschließen. So weiß der Compiler schon vor der Ausführung, welchen erforderlichen Datentyp dieser für den Rückgabewert und den Parameter bearbeiten muss. Mehr Informationen sind für ihn zunächst nicht nötig. Die Definition der Funktion kann sich dann an einer beliebigen anderen Stelle (gewöhnlich in einer anderen Quelldatei) befinden. Eine solche Deklaration wird auch als Vorausdeklaration (*Forward*-Deklaration) bezeichnet.

Hierzu nochmals das Listing `listing002.c`, nur dieses Mal mit einer nötigen Vorausdeklaration in der Zeile **(02)**, wie es eben beschrieben wurde:

```
00  // kap007/listing003.c
01  #include <stdio.h>

02  void hallo(void);  // Funktionsprototyp

03  int main(void) {
04    printf("Vor der Funktion\n");
05    hallo();  // Funktionsaufruf
```

```
06    printf("Nach der Funktion\n");
07    return 0;
08  }

09  // Funktionsdefinition
10  void hallo(void) {
11    printf("In der Funktion\n");
12  }
```

Natürlich ist der Zweck einer Vorausdeklaration eines solchen Funktionsprototypen wie in der Zeile **(02)** nicht der, eine Funktion im selben Quellcode hinter der `main`-Funktion zu definieren. In der Praxis werden Sie solche Deklarationen von Funktionsprototypen eher dazu verwenden, wenn es darum geht, ihren Quellcode in mehreren einzelnen Modulen auf unterschiedlichen Quellcodes aufzuteilen. Hierbei wird häufig die Deklaration (Funktionsprototyp) wie in der Zeile **(02)** und die Definition wie in der Zeile **(09)** bis **(12)** jeweils in eine separate Header- und Quelldatei aufgeteilt.

7.4 Funktionsparameter

Funktionen ohne Parameter mit dem Schlüsselwort `void` kennen Sie bereits. Sie können aber auch Werte an eine Funktion übergeben. Meist stammen die Werte, die Sie übergeben möchten, aus Variablen. Hierbei haben Sie zwei Möglichkeiten: Sie kopieren den aktuellen Wert des Argumentes in den formalen Parameter der Funktion. Oder Sie kopieren lediglich die Adresse des Argumentes in den formalen Parameter der Funktion. In diesem Abschnitt geht es zunächst nur um die Übergabe als Kopie.

Im Folgenden sehen Sie ein einfaches Beispiel, wie Sie Werte an eine Funktion übergeben und damit weiterarbeiten:

```
00  // kap007/listing004.c
01  #include <stdio.h>

02  void multi(int ival1, int ival2) {
03    printf("%d * %d = %d\n", ival1, ival2, ival1*ival2);
04  }
```

```
05  int main(void) {
06    int val1 = 0, val2 = 0;
07    printf("Bitte zwei positive Ganzzahlen: ");
08    if( scanf("%d %d", &val1, &val2) != 2 ) {
09      printf("Fehler bei der Eingabe...\n");
10      return 1;
11    }
12    multi( val1, val2 );
13    return 0;
14  }
```

In der Funktionsdefinition der Zeile **(02)** wird die formale Parameterliste festgelegt. Im Beispiel wurden zwei Integer-Werte vereinbart. Nach dem in der `main`-Funktion zwei Integer-Werte eingelesen wurden, wird in Zeile **(12)** die Funktion mit den Argumenten aufgerufen. Jetzt kann die Funktion mit den Zeilen **(02)** bis **(04)** mit den Parametern arbeiten. Im Beispiel wird nur ausgegeben, wie viel die Multiplikation der beiden Integer-Werte ergibt.

Formale Parameterliste

Besonders wichtig, wenn Sie eine Funktion mit den Parametern (auch formale Parameterliste genannt) aufrufen, ist, dass diese Argumente mit dem Typ und der Reihenfolge des formalen Parameters übereinstimmen müssen. Sollten Sie beispielsweise versehentlich oder absichtlich einen falschen Datentyp als Argument an die Funktion übergeben, findet eine implizite Typumwandlung (sofern keine explizite Umwandlung erfolgt ist) statt.

Das Programm bei der Ausführung:

```
Bitte zwei positive Ganzzahlen: 10 3
10 * 3 = 30
```

Beachten Sie, dass beim *call-by-value* immer eine Kopie vom Speicherobjekt des Aufrufers an die Funktion übergeben wird. Eine Veränderung der Werte in der Funktion hat keine Auswirkung auf die Werte des Aufrufers.

Des Weiteren benötigt eine Werteübergabe als Kopie mehr Rechenleistung, weil die Werte bei der Übergabe extra kopiert werden müssen.

Implizite Datentypenumwandlung

Wenn Sie nicht den Datentypen als Argument an die Funktion übergeben, welcher als formaler Parameter vereinbart wurde, findet eine implizite Typenumwandlung statt (sofern Sie keine explizite Umwandlung durchführen). Hierbei gelten dieselben Regeln, wie dies schon in Abschnitt 4.6, »Implizite Typumwandlung«, umfangreich beschrieben wurde. Hierzu ein Beispiel:

```
void circle(int radius) {
   // Anweisungen
}

int main(void) {
  double dval = 3.33;
  circle(dval);
  ...
}
```

In diesem Beispiel wird zwar ein Gleitpunktyp an die Funktion `circle()` als Argument übergeben, aber der formale Parameter von `circle()` ist ein Integer-Wert. Daher wird eine implizite Umwandlung von `float` nach `int` durchgeführt und die Nachkommastelle abgeschnitten.

7.5 Rückgabewert von Funktionen

In der Praxis werden Sie sehr oft Funktionen erstellen, die eine Berechnung oder andere Arbeiten durchführen. In seltenen Fällen wird eine Funktion keinen Wert (`void`-Funktion) zurückgeben. Entweder gibt eine Funktion den Rückgabewert einer bestimmten Berechnung zurück, oder sie gibt an, ob die Ausführung der Funktion erfolgreich bzw. fehlerhaft war.

Hierzu ein einfaches Beispiel: eine Funktion, die zwei Integer-Werte miteinander vergleicht und den größeren der beiden Werte zurückgibt. Sind beide Zahlen gleich, wird 0 zurückgegeben:

```
00 // kap007/listing005.c
01 #include <stdio.h>

02 int intcmp(unsigned int val1, unsigned int val2) {
03   if( val1 > val2 ) {
04     return val1;
05   }
06   else if (val1 < val2) {
07     return val2;
08   }
09   return 0; // Beide sind gleich.
10 }

11 int main(void) {
12   unsigned int ival1 = 0, ival2 = 0;
13   printf("Bitte zwei Ganzzahlen eingeben: ");
14   if( scanf("%u %u", &ival1, &ival2) != 2 ) {
15     printf("Fehler beider Eingabe...\n");
16     return 1;
17   }
18   int cmp = intcmp( ival1, ival2 );
19   if(cmp != 0) {
20     printf("%d ist der hoehere Wert\n", cmp);
21   }
22   else {
23     printf("Beide Werte sind gleich\n");
24   }
25   return 0;
26 }
```

Anhand des Funktionskopfes der Funktionsdefinition in Zeile **(02)** können Sie durch das Voranstellen des Typs erkennen, dass der Rückgabewert dieser Funktion ein Integer vom Typ `int` ist. Als Parameter erwartet diese Funktion zwei Variablen vom Typen `int`. Aufgerufen wird sie mit den zwei Argumenten in Zeile **(18)**. Damit der Rückgabewert der Funktion auch irgendwo gespeichert wird, wird der Rückgabewert mithilfe des Zuweisungsoperators der `int`-Variable `cmp` zugewiesen.

Welcher Wert an die Variable cmp in Zeile **(18)** übergeben wird, entscheidet sich in der Funktion zwischen den Zeilen **(02)** bis **(10),** wo die größere der beiden Variablen ermittelt wird. Damit am Ende auch etwas aus der Funktion zurückgegeben werden kann, müssen Sie die return-Anweisung in der Funktion dazu verwenden. Ist etwa val1 größer als val2, wird mittels return in Zeile **(04)** die Variable val1 aus der Funktion zurückgegeben und der Variable cmp in Zeile **(18)** zugewiesen. Gleiches geschieht in Zeile **(07)**, wenn val2 größer als val1 ist. In Zeile **(09)** wird 0 zurückgegeben, wenn beide Werte gleich sind.

Das Programm bei der Ausführung:

```
Bitte zwei Ganzzahlen eingeben: 124 125
125 ist der hoehere Wert

Bitte zwei Ganzzahlen eingeben: 5 0
5 ist der hoehere Wert

Bitte zwei Ganzzahlen eingeben: 1 1
Beide Werte sind gleich
```

return-Anweisung

Von großer Bedeutung ist die return-Anweisung in Funktionen. Mit einer return-Anweisung wird eine Funktion beendet und kehrt zum Aufrufer der Funktion zurück und das Programm wird hinter dem Aufrufer weitergeführt. Wird die return-Anweisung hingegen in der main-Funktion verwendet, wie Sie dies in vielen der vorausgehenden Beispiele bei einer fehlerhaften Eingabe von scanf verwendet haben, wird das Programm beendet. Die return-Anweisungen können Sie auf unterschiedliche Arten verwenden. Bei einer Funktion mit dem Rückgabetypen void darf zwar kein Rückgabewert verwendet werden, aber Sie können trotzdem diese Funktion mit einer leeren return-Anweisung vorzeitig beenden:

```
return;  // Funktion mit Rückgabewert void verlassen
```

Es sollte aber hier nicht der Eindruck entstehen, eine Funktion mit dem Rückgabewert void müsse mit return beendet werden. Eine Funktion mit

`void` als Rückgabewert wird auch beendet, wenn das Ende des Funktionskörpers mit den geschweiften Klammern erreicht wird.

Wenn der Rückgabetyp allerdings kein `void` ist, müssen Sie mit `return` einen Wert zurückliefern, der auch ein Ausdruck sein kann:

```
return ausdruck;
```

Ist der Typ von `ausdruck` unterschiedlich zum erwarteten Typ des Aufrufers, der einer Variablen zugewiesen wird, findet eine implizite Umwandlung statt, wie Sie diese von Abschnitt 4.6, »Implizite Typumwandlung«, her kennen.

Funktionsprototypen

Nun, da Sie Funktionen mit Rückgabewert und formale Parameter kennen, soll Abschnitt 7.3, »Funktionsdeklaration (Vorausdeklaration)«, etwas ergänzt werden. Dies betrifft die Art und Weise, wie Sie Funktionsprototypen für die Vorausdeklaration verwenden dürfen. Bisher kennen Sie ja die folgende Deklaration eines Funktionsprototyps:

```
int tmpval(float val1, int val2);

int main(void) {
  int val = tmpval(3.23, 10);
  // ...
  return 0;
}

int tmpval(float val1, int val2) {
  // ...
  return val2;
}
```

Verwenden Sie den Prototyp bei der Vorausdeklaration, müssen Sie allerdings nicht zwangsläufig die Parameter-Bezeichner dafür nutzen. Somit können Sie die Deklaration auch wie folgt angeben:

```
// ohne Bezeichner auch erlaubt
int tmpval(float, int);
```

7.6 Exkurs: Funktion bei der Ausführung

Wenn Sie eine Funktion aufrufen, werden einige Mechanismen in Gang gesetzt, die einen reibungslosen Ablauf gewährleisten. Gerade bei der Verwendung von lokalen Variablen und der Rückgabe von Werten aus Funktionen ist es hilfreich zu wissen, was hinter den Kulissen beim Funktionsaufruf geschieht.

Rufen Sie eine Funktion auf, müssen einige Daten für die Funktion gespeichert werden und auch Daten, um nach dem Ende der Funktion wieder mit der Ausführung hinter dem Funktionsaufruf fortzufahren. Wird also ein Funktionsaufruf gestartet, wird auf dem Stackrahmen (Stacksegment) Speicherplatz dafür reserviert.

Jedes Programm hat einen solchen Stackrahmen, wo bei Bedarf Speicher reserviert und wieder freigegeben wird. Auf dem Stack wird also für jeden Funktionsaufruf ein Datenblock angelegt. In diesem Datenblock werden die formalen Parameter, die lokalen Variablen und die Rücksprungadresse zum Aufrufer gespeichert.

Wird eine Funktion mit `return` beendet, oder erreicht die Funktion das Ende des Bezugsrahmens (Anweisungsblock), werden diese Daten im Stackrahmen wieder freigegeben. Das bedeutet auch, dass alle lokalen Variablen, die in der Funktion definiert wurden und deren Wert nach dem Ende der Funktion ebenfalls verloren sind, freigegeben werden.

Wenn Sie in einer Funktion eine weitere Funktion aufrufen, wird ein zusätzlicher Datenblock unter dem Datenblock der aktuellen Funktion angelegt.

7.7 Inline-Funktionen

Wenn Sie eine Funktion in C mit dem Schlüsselwort `inline` definieren, dann soll diese möglichst »schnell« aufgerufen werden. Dies kann beispielsweise dadurch erfolgen, dass diese Funktion nicht mehr als eigenständiger Code auf dem Stack gelegt wird und extra aufgerufen werden muss, sondern an der entsprechenden Stelle des Funktionsaufrufes direkt eingefügt wird.

Hierzu zunächst ein Programmbeispiel:

```
00 // kap007/listing006.c
01 #include <stdio.h>

02 inline double kreisflaeche(double r) {
03   return  3.14159265358979323846 * r * r;
04 }

05 int main(void) {
06   double radius = 1.0, flaeche = 0.0;
07   for(int i = 0; i < 100; i++) {
08     flaeche = kreisflaeche(radius);
09     printf("Radius: %lf Flaeche: %lf\n", radius, flaeche);
10     radius += 1.0;
11   }
12   return 0;
13 }
```

Die Funktion `kreisflache()` in den Zeilen **(02)** bis **(04)** wurde mit dem Schlüsselwort `inline` versehen. Mit diesem weisen Sie den Compiler an, den Code der `inline`-Funktion möglichst schnell auszuführen. Eine solche Optimierung könnte beispielsweise lauten, den Code direkt an der Stelle einzufügen (engl. *Inline Substitution*), wo diese Funktion aufgerufen wird. Im Beispiel *könnte* somit der Code der Funktion in der Zeile **(08)** eingefügt, wo diese in einer Schleife 100x aufgerufen wird.

Der Compiler kann ein `inline` allerdings auch ignorieren und diese Funktion wie eine normale Funktion behandeln. Mit dem Schlüsselwort `inline` schlagen Sie dem Compiler nur vor, diese Funktion doch *inline* zu verwenden. Welche Funktionen der Compiler als *inline* behandelt, entscheidet er letztendlich selbst. In der Praxis sollten Sie `inline`-Funktionen möglichst klein halten. Dann stehen die Chancen gut, dass der Compiler sie auch tatsächlich *inline* in das Programm einbaut. Wie genau das Schlüsselwort `inline` vom Compiler behandelt wird, hängt von seiner Implementierung ab.

7.8 Rekursionen

Eine Rekursion ist eine Funktion, die sich selbst aufruft. Damit sich aber eine Rekursion nicht unendlich oft selbst aufruft, sondern irgendwann auch zu einem Ergebnis kommt, benötigen Sie wie bei einer Schleife eine Abbruchbedingung. Sonst kann es passieren, dass Ihr Computer abstürzt, da eine Funktion, die sich immer wieder selbst aufruft, eine Rücksprungadresse, den Wert der Variablen und – falls noch nicht freigegeben – den Rückgabewert speichert. Der dafür zur Verfügung stehende Speicher (Stacksegment) wird irgendwann voll sein beziehungsweise überlaufen (auch genannt Stacküberlauf oder *Stack Overflow*).

Hierzu ein einfaches Beispiel, womit die Fakultät der Zahl *n* berechnet wird. Die Fakultät der Zahl 5 ist z. B.: 1×2×3×4×5 = 120

```
00 // kap007/listing007.c
01 #include <stdio.h>

02 long fakul( long n ) {
03   if(n != 0 ) {
04     return n * (fakul(n-1));
05   }
06   return 1;
07 }

08 int main(void) {
09   printf("Fakultaet von 6: %ld\n", fakul(6));
10   printf("Fakultaet von 8: %ld\n", fakul(8));
11   return 0;
12 }
```

Zunächst wird die Funktion aus der `main`-Funktion zweimal aufgerufen. Einmal mit der Zahl 6 (Zeile **(09)**) und einmal mit der Zahl 8 (Zeile **(10)**). In der Funktion wird zunächst überprüft (Zeile **(03)**), ob der Wert von `n` ungleich 0 ist. Diese Überprüfung ist auch gleichzeitig die Abbruchbedingung für die Rekursion, weil eine Multiplikation mit 0 (`n*0`) das Ergebnis 0 ergeben würde. In der Zeile **(04)** wird der aktuelle `n`-Wert mit dem Rückgabewert eines erneuten rekursiven Funktionsaufrufs von `fakul()` multipli-

ziert. Als Argument wird `n` um eins reduziert. Bei einer Fakultät von 6 wäre dies somit 6×5×4×3×2×1. Bei einem erneuten rekursiven Aufruf mit `fakul(0)` greift die Abbruchbedingung der Zeile **(03)**, dass `n!=0` nicht mehr zutrifft. Es wird 1 mit `return` (Zeile **(06)**) zurückgegeben. Im Stack wird jetzt Funktion für Funktion von unten nach oben bis zur `main`-Funktion zurückgesprungen.

Das Programm bei der Ausführung:

```
Fakultät von 6: 720
Fakultät von 8: 40320
```

An dieser Stelle sei gesagt, dass dieses Beispiel auch ohne Rekursion programmiert werden kann. Aber dies soll Teil einer Übungsaufgabe am Ende des Kapitels werden. Des Weiteren muss erwähnt werden, dass sich fast alle Probleme (Algorithmen) auch ohne Rekursionen bewerkstelligen lassen. Viele verschiedene Algorithmen lassen sich aber wesentlich einfacher mit Rekursionen formulieren und programmieren. Meistens werden z. B. rekursive Funktionen beim Durchlaufen von baumartigen Strukturen (Stichwort: binäre Bäume, binäre Suche usw.) verwendet. Obgleich es auch dafür häufig eine iterative Lösung gibt.

7.9 main-Funktion

Die `main()`-Funktion lautet seit dem C99-Standard richtig:

```
int main(void) {
  return 0; // muss nicht verwendet werden
}
```

Weiterhin ist auch eine Variante mit zwei Parametern erlaubt:

```
int main(int argc, char *argv[]) {
  return 0; // muss nicht verwendet werden
}
```

Mehr zu dieser Schreibweise mit den Argumenten `argc` und `argv` erfahren Sie in Abschnitt A.4, »Kommandozeilenargumente«.

Rückgabewert an das Betriebssystem

Der Rückgabewert beim Beenden eines Programms ist gewöhnlich 0, wenn das Programm erfolgreich beendet wurde; alles andere bedeutet eben, dass etwas schiefgelaufen ist.

Zur Vereinfachung finden Sie mit den Makros `EXIT_SUCCESS` und `EXIT_FAILURE` einen recht zuverlässigen Weg, um ein Programm zu beenden. Beide sind in der Headerdatei `<stdlib.h>` definiert. Damit müssen Sie sich nicht mehr selbst darum kümmern, welchen Wert auf welchem System Sie zurückgeben müssen, um zu melden, ob sich eine Anwendung erfolgreich oder eben nicht erfolgreich beendet hat. Bei einem erfolgreichen Ende geben Sie einfach `EXIT_SUCCESS` zurück und bei einem Fehler `EXIT_FAILURE`. Natürlich müssen Sie hierfür auch die Headerdatei `<stdlib.h>` mit einbinden.

Impliziter Rückgabewert aus »main«

Der Funktionsblock von `main()` muss nicht unbedingt eine `return`-Anweisung erhalten. Wird das Blockende von `main()` erreicht, wird implizit 0 als Rückgabewert zurückgegeben.

Ein einfaches Beispiel zum Beendigungsstatus `EXIT_SUCCESS` und zu `EXIT_FAILURE`:

```
00  // kap007/listing008.c
01  #include <stdio.h>
02  #include <stdlib.h>

03  int main(void) {
04    int ival = 0;
05    printf("Ganzzahl eingeben: ");
06    if(scanf("%d", &ival) != 1) {
07      printf("Fehler bei der Eingabe!\n");
08      return EXIT_FAILURE;
09    }
10    return EXIT_SUCCESS;
11  }
```

In diesem Beispiel wird in Zeile **(06)** eine Ganzzahl vom Typ `int` eingelesen und überprüft, ob `scanf` einen Wert ungleich 1 zurückgegeben hat. Das bedeutet, dass kein gültiger (`int`-)Wert eingelesen bzw. umgewandelt werden konnte. Es wird eine Fehlermeldung ausgegeben und das Programm mit dem `return`-Wert `EXIT_FAILURE` beendet. Beim normalen Ablauf wird das Programm mit `EXIT_SUCCESS` in der Zeile **(10)** beendet.

Das Programm bei der Ausführung:

```
$ ./listing008
Ganzzahl eingeben: 5
$ echo $?
0
$ ./listing008
Ganzzahl eingeben: x
Fehler bei der Eingabe!
$ echo $?
1
```

Bei der Ausführung des Beispiels wurde außerdem demonstriert, wie Sie den Rückgabewert in einem Linux-, OS X- oder einem beliebigen UNIX-Terminal mithilfe der Shellvariablen `$?` auswerten können. Diese Variable enthält den Rückgabewert des zuletzt ausgeführten Kommandos. Unter MS Windows-Systemen können Sie den Errorlevel beispielsweise im Batch-Modus auswerten.

7.10 Programm mit exit() beenden

Wollen Sie ein Programm an einer beliebigen Position beenden, können Sie die Funktion `exit()` aus der Headerdatei `<stdlib.h>` dazu verwenden. Diese Funktion dient dazu, ein Programm ordentlich mit einem Statuswert zwischen den Klammern zu beenden. Auch hier gilt wie schon bei der Rückgabe mittels `return` aus der `main`-Funktion, dass in der Regel der Wert 0 verwendet wird, wenn ein Programm ordentlich beendet wurde. Daher bietet es sich hier auch hier an, die Makros `EXIT_SUCCESS` und `EXIT_FAILURE` zu verwenden, um eine erfolgreiche bzw. fehlerhafte Beendigung des Programms zurückzugeben.

Ein einfaches Beispiel hierzu:

```
00 // kap007/listing009.c
01 #include <stdio.h>
02 #include <stdlib.h>

03 int readint(void) {
04   int val = 0;
05   printf("Bitte eine Ganzzahl eingeben: ");
06   if(scanf("%d", &val) != 1) {
07     printf("Fehler bei der Eingabe!\n");
08     exit(EXIT_FAILURE);
09   }
10   return val;
11 }

12 int main(void) {
13   int ival = readint();
14   printf("Der Wert lautet: %d\n", ival);
15   return EXIT_SUCCESS;
16 }
```

Das Beispiel entspricht im Grunde dem Listing *listing008.c* nur wurde hier das Einlesen des `int`-Wertes als Funktion in der Zeile **(03)** bis **(11)** geschrieben. Wenn `scanf` den einen Wert nicht korrekt einlesen konnte, beendet die Funktion `readint()` in der Zeile **(08)** das komplette Programm mithilfe der Funktion `exit()`.

Wenn Sie die `main`-Funktion mit `exit()` beenden, ist dies gleichwertig zur Beendigung mit einem `return` innerhalb von `main`.

Funktionen ohne Wiederkehr (seit C11)

Einige Funktionen kehren aus verschiedenen Gründen nicht mehr zu ihrem Aufrufer zurück und beenden das Programm. Die Funktionen `exit()` oder `abort()` sind zwei solche Vertreter aus der Standardbibliothek. Oftmals werden auch eigene *Exit*-Funktionen geschrieben, um vor der Beendigung beispielsweise noch. Aufräumarbeiten durchzuführ-

ren. Mit dem neuen Schlüsselwort `_Noreturn` können Sie seit C11 selbst eine solche Funktion wie `exit()` oder `abort()` markieren, die nicht mehr zu ihrem Aufrufer zurückkehrt. Der Vorteil dieses Schlüsselworts liegt darin, dass der Compiler mit diesem Wissen einen besseren Code erstellen kann und so auch eine sinnvollere Aussage von Analysewerkzeugen erwartet werden kann. Zum Beispiel:

```
_Noreturn void function(); // C11, kehrt niemals zurück
```

7.11 Globale, lokale und statische Variablen

In der üblichen Programmierung werden viele Variablen in Funktionen oder in Anweisungsblöcken verwendet. Dabei sollten Sie auf jeden Fall mit dem Bezugsrahmen von Variablen vertraut sein. Grundlegend unterscheidet man zwischen einem *lokalen* und einem *globalen Bezugsrahmen*.

7.11.1 Lokale Variablen

Gibt es sowohl eine globale als auch eine lokale Variable mit demselben Bezeichner, erhält bei der Verwendung die Variable den Zuschlag, die bei der Ausführung im engsten Sinne lokal ist. Zum Beispiel gewinnt eine Variable aus demselben innersten Codeblock gegen eine, die lediglich in derselben Funktion oder in einem weiter außen gelegenen Codeblock deklariert wurde. Folgender Codeausschnitt soll diese zeigen:

```
00 // kap007/listing010.c
01 #include <stdio.h>

02 int main(void) {
03   int ival = 123;
04   printf("ival: %d\n", ival);
05   { // Neuer Anweisungsblock
06     ival = 321;
07     printf("ival: %d\n", ival);
08   }
09   printf("ival: %d\n", ival);
```

```
10    { // Neuer Anweisungsblock
11      int ival = 456;
12      printf("ival: %d\n", ival);
13    }
14    printf("ival: %d\n", ival);
15    return 0;
16  }
```

Hier das Programm bei der Ausführung:

```
ival: 123
ival: 321
ival: 321
ival: 456
ival: 321
```

Sie weisen zunächst in Zeile **(03)** den Wert 123 an die Variable `ival` zu, was die Ausgabe in der Zeile **(04)** dann auch ausgibt. In den Zeilen **(05)** bis **(08)** wird ein neuer Anweisungsblock verwendet, in dem in Zeile **(06)** der Wert von `ival` auf 321 geändert wird. Das bestätigt die Ausgabe in Zeile **(07)**. Außerhalb des Anweisungsblocks in Zeile **(09)** wird ebenfalls der Wert 321 ausgegeben. In den Zeilen **(10)** bis **(13)** wird nochmals ein Anweisungsblock verwendet, nur wird jetzt in der Zeile **(11)** eine neue lokale Variable `ival` definiert und mit dem Wert 456 versehen. Das bestätigt die Ausgabe der Zeile **(12)**. Die in Zeile **(11)** definierte Variable ist allerdings nur innerhalb des Anweisungsblocks gültig und überdeckt in diesem Block die gleichnamige Variable außerhalb des Anweisungsblocks. Dieses bestätigt wieder die Ausgabe der Zeile **(14)**. Dort ist der Wert von `ival` nach wie vor 321. Zwar wurde in diesem Beispiel nur die `main`-Funktion verwendet, aber das Prinzip der lokalen Variablen ist allgemeingültig. Einige Compiler geben hierbei auch eine Warnmeldung dieser gleichnamigen Überdeckung in der Zeile **(11)** aus.

Regel für lokale Variablen

Somit gilt für lokale Variablen Folgendes: Bei gleichnamigen Variablen innerhalb derselben Funktion ist immer die »lokalste« Variable gültig,

also diejenige, die dem Anweisungsblock am nächsten steht. In der Praxis wird man allerdings eher selten hergehen und gleichnamige Variablen in derselben Funktion verwenden. Das Beispiel dient hier nur dem Zweck, den lokalen Bezugsrahmen zu demonstrieren.

7.11.2 Globale Variablen

Während Sie mit lokalen Variablen auf die Funktionen bzw. auf den Anweisungsblock selbst beschränkt sind, können Sie globale Variablen in allen Funktionen verwenden. Hierzu zunächst ein einfaches Beispiel mit einer globalen Variable, die in Zeile (02) definiert wurde:

```
00 // kap007/listing011.c
01 #include <stdio.h>
02 int ival=789;

03 void funktion1(void) {
04   printf("ival: %d\n", ival);
05 }

06 void funktion2(void) {
07   int ival = 111;
08   printf("ival: %d\n", ival);
09 }

10 int main(void) {
11   int ival = 123;
12   printf("ival: %d\n", ival);
13   { // Neuer Anweisungsblock
14     ival = 321;
15     printf("ival: %d\n", ival);
16   }
17   printf("ival: %d\n", ival);
18   { // Neuer Anweisungsblock
19     int ival = 456;
```

```
20      printf("ival: %d\n", ival);
21    }
22    printf("ival: %d\n", ival);
23    funktion1();
24    funktion2();
25    return 0;
26  }
```

Das Programm bei der Ausführung:

```
ival: 123
ival: 321
ival: 321
ival: 456
ival: 321
ival: 789
ival: 111
```

Die Ausgabe des Programms dürfte vielleicht den einen oder anderen etwas überraschen. Obwohl in Zeile **(02)** eine globale Variable `ival` definiert wurde, kommt sie nur einmal in der Funktion `funktion1()` in der Zeile **(03)** bis **(05)** zum Einsatz. Ansonsten entspricht die Ausführung dem Listing *listing010.c*, nur dass hier noch eine Funktion `funktion2()` definiert (Zeile **(06)** bis **(09)**) wurde. Sie verwendet allerdings auch nur eine darin definierte Variable `ival` (Zeile **(07)**).

Auch wenn eine globale Variable definiert wird und eine gleichnamige lokale Variable existiert, erhält immer auch hier die lokalste Variable (die dem Anweisungsblock am nächsten steht) den Zuschlag und überdeckt die Variable die außerhalb des Blockes steht. Existieren in einem Programm unwissentlich zwei gleichnamige Variablen, nämlich eine globale und eine lokale, kann es zu unerwarteten Ergebnisse kommen. Die Lebensdauer der globalen Variable erstreckt sich bis zum Programmende.

Vermeiden Sie globale Variablen

Da globale Variablen für alle Funktionen sichtbar sind, die nach diesen Variablen in einer Datei definiert werden, sollten Sie die Verwendung

dieser nochmals überdenken, ob es nicht ohne auch geht. Wenn mehrere Funktionen auf eine globale Variable zugreifen, so kann es hier schnell mal zu einer fehlerhaften Veränderung des Wertes kommen. Dabei wird es dann häufig sehr schwierig, den Fehler bzw. die Funktion, welche den Fehler verursacht hat, zu lokalisieren.

Globale Variablen werden außerdem, im Gegensatz zu den lokalen Variablen, bei der Definition automatisch mit 0 initialisiert.

7.11.3 Speicherklasse»static«

Wenn bei lokalen bzw. globalen Variablen die Speicherklasse `static` verwenden, wird die Variable im Datensegment mit einer festen Speicheradresse versehen, die bis zum Programmende erhalten bleibt. Der Bezugsrahmen hängt davon ab, ob Sie die Speicherklasse in einem lokalen oder globalen Bereich vereinbaren.

Lokale »static«-Variable innerhalb einer Funktion

Verwenden Sie eine `static`-Variable lokal innerhalb einer Funktion, bleibt der Wert der lokalen `static`-Variablen nach der Rückkehr aus dieser Funktion erhalten. Trotzdem können Sie diese `static`-Variable nur lokal innerhalb derjenigen Funktion ansprechen, in der sie erstellt wurde. Ein einfaches Programmbeispiel hierzu, in dem eine Variable mit dem Speicherklassen-Spezifizierer `static` gekennzeichnet wurde:

```
00 // kap007/listing012.c
01 #include <stdio.h>

02 void counter(void) {
03   static int ival;
04   printf("ival: %d\n", ival);
05   ++ival;
06 }

07 int main(void) {
08   counter();
```

```
09    counter();
10    counter();
11    return 0;
12  }
```

Die Ausgabe des Listings:

```
ival = 0
ival = 1
ival = 2
```

Dank `static` in Zeile **(03)** werden in der Funktion die Werte 0, 1 und 2 ausgegeben, obwohl diese Funktion dreimal hintereinander neu aufgerufen wurde. Daher können Sie sich darauf verlassen, dass bei Beendigung einer Funktion eine statische Variable **nicht** den Wert verliert. Das liegt daran, dass statische Variablen in einer Funktion nicht wie üblich im Stack-Segment gespeichert werden, sondern mit einer festen Speicheradresse im Datensegment. Dadurch hat die Variable eine Lebensdauer bis zum Programmende. Ohne `static` hätte die Variable nur eine Gültigkeit innerhalb der Funktion, sowie eine Lebenszeit während der Laufzeit der Funktion. Des Weiteren können Sie sicher sein, wie das Beispiel mit der Ausgabe auch zeigt, dass eine `static`-Variable automatisch mit 0 initialisiert wird.

»static«-Variable außerhalb einer Funktion

Deklarieren Sie hingegen eine `static`-Variable außerhalb einer Funktion, ist sie nur in der Datei gültig, wo sie deklariert wurde – sprich, die Variable kann von keiner anderen Programmdatei verwendet werden. Innerhalb der Programmdatei können Sie diese Variable wie eine gewöhnlich globale Variable verwenden. Haben Sie beispielsweise in zwei Quelldateien, die zu einem Programm gehören, die folgende Variable verwendet,

```
// datei-01.c
int ival = 12345;

// datei-02.c
int ival = 34567;
```

beschwert sich der Linker, dass die globale Variable `ival` mehr als nur einmal im Programm definiert wurde. Stellen Sie jetzt in jeder der beiden Quelldateien das Schlüsselwort `static` davor:

```
// datei-01.c
static int ival = 12345;

// datei-02.c
static int ival = 34567;
```

übersetzt der Linker das Programm anstandslos, weil jetzt diese globalen `static`-Variablen nur in der jeweiligen Quelldatei gültig und sichtbar sind. Allerdings soll ein solch gezeigtes Beispiel mehrere Definitionen mit dem gleichen Bezeichner keine Schule machen, sondern es dient auch hier nur zur Demonstration von `static`. Vielmehr sollten Sie globale Variablen, die nicht von anderen Modulen verwendet werden sollen, davor schützen, indem Sie sie mit dem Schlüsselwort deklarieren.

»static«-Funktionen

Dasselbe wie eben für eine globale `static`-Variable gilt für `static`-Funktionen. Auch Funktionen, die Sie mit `static` kennzeichnen, sind nur in der aktuellen Quelldatei gültig, in der sie definiert wurden. So können Sie durchaus zwei Funktionen mit denselben Bezeichnern in unterschiedlichen Quelldateien verwenden, wenn Sie mindestens eine davon mit `static` spezifiziert haben. Auf diese Weise können Sie beispielsweise auch verhindern, dass eine Funktion von einer anderen Datei aufgerufen wird und eben nur innerhalb einer Datei verwendet werden kann.

7.11.4 Die Speicherklasse extern

Bezogen auf globale Variablen dürfte dann hier auch noch die Speicherklasse `extern` von Bedeutung sein. Mit dem Schlüsselwort `extern` gibt der Compiler dem Linker Bescheid, dass dieser die Verweise in einer anderen Übersetzungseinheit (Quell- oder Headerdatei) und/oder Bibliothek auf-

lösen muss. Sie Deklarieren hiermit eine Variable nur und die Definition erstellen Sie gewöhnlich in einer anderen Datei.

Verwenden Sie bei einer Funktionsdeklaration (Prototyp) nicht den Speicherklassen-Spezifizierer `extern`, wird sie implizit automatisch vom Compiler so behandelt, als enthielte sie das Schlüsselwort `extern`. Somit müssen Sie zwar nicht zwangsläufig bei einer Funktionsdeklaration das Schlüsselwort `extern` verwenden, aber es ist auch nicht falsch, weil Sie hiermit immer eine Deklaration erzwingen und es nie eine Definition wird, wo der Compiler Speicherplatz reserviert.

7.12 Kontrollfragen und Aufgaben

1. Was ist eine Vorausdeklaration?
2. Versuchen Sie, den Begriff call-by-value etwas ausführlicher zu beschreiben.
3. Was müssen Sie bei der Verwendung eines Rückgabewertes beachten, und mit welcher Anweisung können Sie einen Wert aus einer Funktion zurückgeben?
4. Welcher Speicher verwaltet bei einem gewöhnlichen Funktionsaufruf (ohne spezielle Speicher-Spezifizierer) die Daten?
5. Was sind Rekursionen?
6. Beschreiben Sie den Unterschied zwischen einer globalen und lokalen Variablen.
7. Was können Sie tun, wenn Sie eine globale Variable verwenden wollen, diese aber nur in der aktuellen Quelldatei und nicht in anderen Quelltexten oder Headerdateien sichtbar sein soll?
8. Wie erhalten Sie innerhalb einer Funktion den Wert einer Variablen, um bei einem erneuten Funktionsaufruf darauf zurückzugreifen?
9. Warum lässt sich das folgende Programm nicht übersetzen?

```
00 // kap007/aufgabe001.c
01 #include <stdio.h>
02 #include <stdlib.h>
```

```
03 int main(void) {
04   float fval = multi(3.33);
05   printf("%.2f\n", fval);
06   return EXIT_SUCCESS;
07 }

08 float multi(float f) {
09   return (f*f);
10 }
```

10. Was wurde bei diesem Beispiel falsch gemacht?

```
00 // kap007/aufgabe002.c
01 #include <stdio.h>
02 #include <stdlib.h>

03 float volumen_Rect(float l, float b, float h) {
04   float volumen = l*b*h;
05 }

06 int main(void) {
07   float vol = volumen_Rect(10.0, 10.0, 12.5);
08   printf("Volumen: %f\n", vol);
09   return EXIT_SUCCESS;
10 }
```

11. Schreiben Sie das Programm bzw. die Funktion `fakul()`, in der Sie mit einer Rekursion die Fakultät von `n` berechnet haben (`kap007/listing007.c`), um, damit die Fakultät ohne Rekursion berechnet wird.

Kapitel 8

Präprozessor-Direktiven

Bevor der Compiler den Quelltext verarbeitet, führt der Präprozessor einen zusätzlichen Übersetzungslauf oder besser Ersetzungslauf durch. Bei Präprozessor-Direktiven steht immer das Zeichen `#` am Anfang der Zeile. Außerdem darf pro Zeile nur eine Direktive eingesetzt werden. Folgendes ist also **nicht** erlaubt:

```
#include <stdio.h> #define MAX_VAL 255
```

Kommentare hingegen dürfen hinter einer Direktive stehen:

```
#include <stdio.h>   // Header für Standardfunktionen
```

Die folgenden Arbeiten fallen für den Präprozessor an:

- String-Literale werden zusammengefasst.
- Zeilenumbrüche mit einem Backslash davor werden entfernt.
- Kommentare werden entfernt und durch Leerzeichen ersetzt.
- Whitespace-Zeichen zwischen Tokens werden gelöscht.

Des Weiteren gibt es Aufgaben für den Präprozessor, die vom Programmierer gesteuert werden können:

- Header- und Quelldateien in den Quelltext kopieren (`#include`)
- symbolische Konstanten einbinden (`#define`)
- bedingte Kompilierung (`#ifdef`, `#elseif`, ...)

8.1 Dateien einfügen mit #include

Die Direktive `#include` kopiert andere, benannte (Include-)Dateien in das Programm. In der Regel handelt es sich dabei um Headerdateien mit der

Dateiendung *.h. Es gibt zwei Möglichkeiten, die Präprozessor-Direktive include zu verwenden:

```
#include <header>
#include "header"
```

Der Präprozessor entfernt die include-Zeile und ersetzt sie durch den Quelltext der include-Datei. Der Compiler erhält anschließend einen modifizierten Text zur Übersetzung.

Natürlich können Sie auch eigene Headerdateien schreiben und diese mit include einkopieren. Haben Sie beispielsweise eine Headerdatei geschrieben und diese im Verzeichnis */user/meinInclude* unter dem Namen meinheader.h gespeichert, dann müssen Sie die Headerdatei am Anfang des Quelltextes mit

```
#include "/user/meinInclude/meinheader.h"
```

einkopieren. Dabei muss das Verzeichnis angegeben werden, in dem die Headerdatei gespeichert wurde.

Dem Compiler Headerdateien mitteilen

Sie sollten nicht den kompletten Pfad zu einer selbst geschriebenen Headerdatei angeben. In der Praxis werden diese Verzeichnisse dem Compiler über spezielle Optionen oder Einstellungen bekannt gemacht. Beim gcc-Compiler können Sie dazu z. B. den Schalter -I in der Kommandozeile verwenden. Bei Entwicklungsumgebungen finden Sie hierfür ebenfalls Optionen bei den Projekteinstellungen.

Schreiben Sie die Headerdatei hingegen zwischen eckige Klammern (wie dies bei Standardbibliotheken meistens der Fall ist), also so:

```
#include <headerdatei.h>
```

dann wird die Headerdatei headerdatei.h im implementierungsdefinierten Pfad gesucht. Dieser Pfad befindet sich in dem Pfad, in dem sich die Headerdateien Ihres Compilers wie beispielsweise stdio.h, stdlib.h, ctype.h, string.h usw. befinden.

Steht die Headerdatei zwischen zwei Hochkommata, also so:

```
#include "headerdatei.h"
```

dann wird sie im aktuellen Arbeitsverzeichnis (Working Directory) oder in dem Verzeichnis gesucht, das mit dem Compileraufruf, beispielsweise mit einer Option wie `-I`, angegeben wurde – vorausgesetzt, Sie übersetzen das Programm in der Kommandozeile oder haben in der Entwicklungsumgebung entsprechende Angaben gemacht. Sollte diese Suche erfolglos sein, wird in denselben Pfaden gesucht, als hätten Sie `#include <datei.h>` angegeben.

8.2 Konstanten und Makros mit #define und #undef

Mit `#define` ist es möglich, Texte anzugeben, die vor der Übersetzung des Programms durch andere Texte ersetzt werden. Auch hier bewirkt das Zeichen `#` vor `define`, dass der Präprozessor zuerst seine Arbeit verrichtet, in dem Fall die Textersetzung. Erst dann wird das werdende Programm vom Compiler in Maschinensprache übersetzt. Die Syntax der `define`-Direktive sieht so aus:

```
#define BEZEICHNER Ersatzbezeichner
#define BEZEICHNER(Bezeichner_Liste) Ersatzbezeichner
```

Bei der ersten Syntaxbeschreibung wird eine symbolische Konstante und im zweiten Fall ein Makro definiert. In der Praxis werden für die symbolische Konstante oder für das Makro gewöhnlich Großbuchstaben verwendet, um sie sofort von normalen Variablen unterscheiden zu können. Dies ist aber keine feste Regel. Für die Namen gelten dieselben Regeln wie schon bei den Bezeichnern (siehe Abschnitt 2.4.1, »Bezeichner«).

8.2.1 Symbolische Konstanten mit #define

Hier ein erstes Programmbeispiel, das eine symbolische Konstante definiert:

```
00  // kap008/listing001.c
01  #include <stdio.h>
```

```
02  #include <stdlib.h>
03  #define WERT 1234

04  int main(void) {
05    int val = WERT * 2;
06    printf("%d\n", WERT);
07    printf("%d\n", val);
08    return EXIT_SUCCESS;
09  }
```

In dem Programm wird jede symbolische Konstante `WERT` mit dem Wert 1234 in der Zeile **(03)** definiert. Wenn Sie das Programm übersetzen, werden vor der Kompilierung alle Namen mit `WERT` im Quelltext (hier in der Zeile **(05)** und **(06))** vom Präprozessor durch 1234 ersetzt.

Die Unantastbaren

Beachten Sie, dass der Compiler selbst nie etwas von der symbolischen Konstante zu sehen bekommt, weil der Präprozessor diese Namen vor dem Compilerlauf durch die entsprechende Konstante ersetzt. Daher sollte Ihnen bewusst sein, dass `#define`-Makros echte Konstanten sind, die Sie zur Laufzeit des Programms nicht mehr ändern können.

Der Vorteil solcher `define`-Konstanten liegt vorwiegend darin, dass das Programm besser lesbar ist und besser gewartet werden kann. Haben Sie z. B. eine symbolische Konstante in einem 10.000-Zeilen-Programm, etwa

```
#define ROW 15
#define COLUMN 80
#define TITLE "Mein Texteditor 0.1"
```

und Sie wollen diese Angaben ändern, müssen Sie nur den Wert bei `define` anpassen. Hätten Sie hier keine symbolische Konstante verwendet, müssten Sie im kompletten Quelltext (der meistens noch in mehrere Quelldateien aufgeteilt ist) nach den Werten `15`, `80` und der Zeichenkette `"Mein Texteditor 0.1"` suchen und diese gegen die neuen Werte ersetzen. Eine

mühevolle und bei Vergessen eines Wertes auch eine fehleranfällige Angelegenheit.

Hierzu ein weiteres Beispiel:

```
00 // kap008/listing002.c
01 #include <stdio.h>
02 #include <stdlib.h>
03 #define PI 3.1415

04 double Kflaeche( double d ) {
05   return d * d * PI / 4;
06 }

07 double Kumfang( double d) {
08   return d * PI;
09 }

10 int main(void) {
11   double d = 0.0;
12   printf("Kreisdurchmesser: ");
13   if( scanf("%lf", &d) != 1 ) {
14     printf("Fehler bei der Eingabe...\n");
15    return EXIT_FAILURE;
16   }
17   printf("Kreisflaeche von %.2f = %.2f\n",d, Kflaeche(d));
18   printf("Kreisumfang von %.2f  = %.2f\n",d, Kumfang(d));
19   return EXIT_SUCCESS;
20 }
```

In diesem Beispiel werden aus einem Kreisdurchmesser die Kreisfläche (Zeilen **(04)** bis **(06)**) und der Kreisumfang (Zeilen **(07)** bis **(09)**) berechnet. Für beide Kreisberechnungen wird die Angabe von *Pi* benötigt. Im Listing wurde `PI` als symbolische Konstante in Zeile **(03)** definiert und in den Berechnungen der Zeilen **(05)** und **(08)** verwendet. Sollten Sie jetzt eine höhere Genauigkeit zu `PI` benötigen, brauchen Sie nur den Wert der Zeile **(03)** ändern. Sie müssen nicht im Programm danach suchen.

Das Programm bei der Ausführung:

```
Kreisdurchmesser: 5.5
Kreisflaeche von 5.50 = 23.76
Kreisumfang von 5.50  = 17.28
```

Natürlich ist es auch möglich, sich eine Konstante errechnen zu lassen. Die Konstante `PI` beispielsweise können Sie auch wie folgt definieren:

```
#include <math.h>   // benötigter Header für atan()
#define PI atan(1)*4
```

Wird das Programm ausgeführt, steht dort jedes Mal `atan(1)*4`. Das führt dazu, dass dieser Wert jedes Mal da, wo `PI` durch die Berechnung ersetzt wurde, erneut berechnet werden muss. Daher sollten Sie für Berechnungen besser eine `const`-Variable verwenden. Mit dieser muss nur einmal gerechnet werden:

```
#include <math.h>   // benötigter Header für atan()
const double PI = atan(1)*4;
```

8.2.2 Makros mit #define

Neben symbolischen Konstanten lassen sich mit der `define`-Direktive auch parametrisierte Makros erstellen. Hierzu ein Beispiel:

```
00  // kap008/listing003.c
01  #include <stdio.h>
02  #include <stdlib.h>
03  #define NOT_EQUAL(x, y)   ((x) != (y))
04  #define XCHANGE(x, y) { \
        int j; j=(x); (x)=(y); (y)=j; }

05  int main(void) {
06    int ival1 = 0, ival2 = 0;
07    printf("Bitte zwei Ganzzahlwerte eingeben: ");
08    if( scanf("%d %d", &ival1, &ival2) != 2 ) {
09      printf("Fehler bei der Eingabe...\n");
10      return EXIT_FAILURE;
```

```
11    }
12    if( NOT_EQUAL(ival1, ival2) ) {
13      XCHANGE(ival1, ival2);
14    }
15    printf("val1: %d - val2: %d\n", ival1, ival2);
16    return EXIT_SUCCESS;
17  }
```

Parametrisierte Makros erkennen Sie daran, dass nach dem Makronamen eine Klammer folgt, wie dies in den Zeilen **(03)** und **(04)** der Fall ist. Mehrere Parameter werden mit einem Komma voneinander getrennt. Beide Makros haben hier die formalen Parameter `x` und `y`. In Zeile **(12)** wird das Makro `NOT_EQUAL` verwendet. Nach dem Präprozessorlauf sieht die Zeile wie folgt aus:

```
12   if( ival1 != ival2 ) {
```

Trifft es bei der Ausführung der Zeile **(12)** zu, dass beide Werte nicht gleich sind, werden die zwei Werte getauscht. Die Zeile **(13)** im Quelltext wird vom Präprozessor wie folgt ersetzt:

```
13     int j; j=ival1; ival1=ival2; ival2=j; }
```

Zeilenumbrüche in Makros müssen Sie wie in Zeile **(04)** mit einem Backslash-Zeichen am Ende herbeiführen, weil sonst das Makro am Zeilenende beendet würde.

Achtung, keine Typenprüfung

An dieser Stelle soll noch ein Warnhinweis für parametrisierte Makros ausgesprochen werden. Bedenken Sie bitte, dass bei solchen Makros keine Typenprüfung stattfindet. So können Sie das Makro `NOT_EQUAL` aus Zeile **(03)** des Beispiels *listing003.c* auch mit anderen Typen wie `float` oder gar Zeigern aufrufen. Wo bei einfachen Datentypen noch die Gefahr besteht, dass einfach nur Murks aus den Werten gemacht wird, kann dies bei Zeigern schwerwiegendere Folgen haben, bis hin zum Absturz des Programms.

Sie sollten sich außerdem angewöhnen, die Parameter im Ersatztext in Klammern zu setzen. Ein einfaches Beispiel verdeutlicht, was sonst passieren kann:

```
00  // kap008/listing004.c
01  #include <stdio.h>
02  #include <stdlib.h>
03  #define MINUS(a, b) (a-b)

04  int main(void) {
05    printf("%f\n", MINUS(5.0, 2.5+0.5) );
06    return EXIT_SUCCESS;
07  }
```

Durch die Ersetzung der Zeile **(03)** in Zeile **(05)** sieht die Berechnung wie folgt aus:

```
5.0-2.5+0.5
```

Als Ergebnis erhalten Sie hier 3.0. Das korrekte Ergebnis wäre allerdings 2.0, weil die Berechnung eigentlich 5.0–(2.5+0.5) lauten müsste.

Ausdruck in Klammern

Wenn der Wert einer Konstante keine einfache Zahl ist, sondern ein Ausdruck, sollten Sie den Parameter im Ersetzungstext immer zwischen Klammern setzen.

Um also wirklich sicherzugehen, dass auch solche Teilausdrücke richtig ersetzt werden, sollten Sie die Parameter im Ersatztext immer in Klammern setzen. In diesem Beispiel sollte die `define`-Anweisung der Zeile **(04)** deshalb folgendermaßen aussehen:

```
04  #define MINUS(a, b) ( (a) - (b) )
```

Funktionen oder Makros

Wenn Sie anhand der letzten Abschnitte ein wenig Zweifel bekommen haben, ob Sie Makros mit Parametern verwenden können, so sind diese berechtigt. Da bei Makros keinerlei Typenprüfung durchgeführt wird und viele weitere Stolperstellen vorhanden sind, sollten Sie immer bevorzugt auf Funktionen zurückgreifen.

8.2.3 Symbolische Konstanten und Makros aufheben (#undef)

Sollen eine symbolische Konstante oder ein Makro ab einer bestimmten Stelle im Programm nicht mehr gültig sein, können Sie sie mit der `undef`-Anweisung wie folgt aufheben:

```
#undef BEZEICHNER
```

Möchten Sie hingegen nach der `undef`-Anweisung erneut auf `BEZEICHNER` zugreifen, beschwert sich der Compiler, dass dieser Bezeichner nicht deklariert wurde.

Entfernen Sie eine Konstante mit `undef`, welche gar nicht existiert, wird diese Anweisung vom Präprozessor ignoriert. Sie hat somit keine Auswirkung auf das Programm.

In der Praxis haben Sie mit `undef` die Möglichkeit, die Gültigkeit für ein mit `define` eingeführtes Makro auf einen bestimmten Programmabschnitt einer Datei zu begrenzen.

8.3 Bedingte Kompilierung

Sie können für den Präprozessor auch Bedingungen formulieren, damit dieser nur dann eine bestimmte symbolische Konstante, ein Makro oder eine Headerdatei verwendet, wenn eine gewisse Bedingung gegeben ist. Folgende Schlüsselwörter stehen Ihnen für die bedingte Kompilierung zu Verfügung (siehe Tabelle 8.1).

Schlüsselwort	Beschreibung
`#if ausdruck`	Wird `ausdruck` erfüllt (ungleich 0), wird der darauffolgende Programmteil ausgeführt.

`#ifdef symbol`
`#if defined symbol`
`#if defined(symbol)`
	Ist `symbol` im Programm definiert, wird der darauffolgende Programmteil ausgeführt.

`#ifndef symbol`
`#if !defined symbol`
`#if !defined(symbol)`
	Ist `symbol` im Programmteil **nicht** definiert, wird der darauffolgende Programmteil ausgeführt.

`#elif ausdruck` `#elif symbol`	Trifft `ausdruck` zu (ungleich 0) oder ist `symbol` im Programmteil definiert, wird der folgende Programmteil ausgeführt. Einem `#elif` geht immer ein `#if` oder ein `#ifdef` voraus.
`#else`	Der alternative Programmteil dahinter wird ausgeführt, wenn ein vorausgehendes `#if`, `#ifdef`, `#ifndef` oder `#elif` nicht ausgeführt wird.
`#endif`	Zeigt das Ende der bedingten Kompilierung an.

Tabelle 8.1 Schlüsselwörter zur bedingten Kompilierung

Hierzu ein einfaches Beispiel zur bedingten Kompilierung:

```
00  // kap008/listing005.c
01  #include <stdio.h>
02  #include <limits.h>
03  #include <stdlib.h>
04  #define TEST

05  int main(void) {
06  #if UINT_MAX == 4294967295
07    unsigned int val1 = 0;
```

```
08    unsigned int val2 = 0;
09  #else
10    unsigned long val1 = 0;
11    unsigned long val2 = 0;
12  #endif
13    val1 = 500 * 1000;
14    val2 = 300 * 200;
15    val1+= val2;
16    printf("%lu\n", (long)val1);
17  #ifdef ULLONG_MAX
18    unsigned long long uval = val1;
19    printf("%llu\n", uval*10000);
20  #else
21    printf("Erweiterte Berechnung nicht moeglich\n");
22  #endif
23  #ifdef TEST
24    printf("UINT_MAX   : %u\n", UINT_MAX);
25    printf("ULONG_MAX  : %lu\n", ULONG_MAX);
26  #endif
27  #if defined(TEST) && defined(ULLONG_MAX)
28    printf("ULLONG_MAX : %llu\n", ULLONG_MAX);
29  #endif
30    return EXIT_SUCCESS;
31  }
```

In der Zeile **(06)** prüft der Präprozessor, ob der maximale Wert von `UINT_MAX` in der Headerdatei `<limits.h>` auf dem System gleich 4294967295 ist. Trifft dies zu, wird `unsigned int` für die Variablen `val1` und `val2` aus den Zeilen **(07)** und **(08)** verwendet. Trifft es hingegen nicht zu, weil das Programm beispielsweise auf einem 16-Bit-System übersetzt wird, wo `UINT_MAX` dem Wert 65535 entspricht, verzweigt der Präprozessor zur alternativen Verzweigung der Zeile **(09)** und verwendet in den Zeilen **(10)** und **(11)** `unsigned long` als Datentyp für `val1` und `val2`. Entsprechend dem gewählten Wert werden diese in den Zeilen **(13)** bis **(16)** verwendet.

In der Zeile **(17)** wird geprüft, ob `ULLONG_MAX` aus der Headerdatei `<limits.h>` definiert ist. Dies ist beispielsweise nicht der Fall, wenn der Code auf einem Compiler läuft, der den C99-Standard nicht beherrscht. Wenn `ULLONG_MAX` definiert ist, wird `unsigned long long` in den Zeilen **(18)** und **(19)** für eine weitere Berechnung verwendet. Der Wert `unsigned long` wäre dafür nicht mehr geeignet gewesen. Ist `ULLONG_MAX` hingegen nicht definiert, wird in der Zeile **(20)** zur Alternative verzweigt, wo mit `printf` eine Meldung auf dem Bildschirm ausgegeben wird, dass eine erweiterte Berechnung nicht möglich ist.

In der Zeile **(23)** wird geprüft, ob eine symbolische Konstante `TEST` definiert wurde, was in diesem Beispiel in der Zeile **(04)** der Fall ist, weshalb die Testausgabe in den Zeilen **(24)** und **(25)** ausgegeben wird. In der Zeile **(27)** wird mit einer UND-Verknüpfung überprüft, ob `TEST` **und** `ULLONG_MAX` in `<limits.h>` definiert sind. Trifft beides zu, wird auch hier die Testausgabe der Zeile **(28)** ausgegeben.

Wollen Sie die Testausgabe nicht mehr ausgeben, müssen Sie entweder nur Zeile **(04)** entfernen oder ein `#undef` vor die erste Verwendung der symbolischen Konstante `TEST` setzen.

Mehrfaches Inkludieren vermeiden

Sie können mit der bedingten Kompilierung auch überprüfen, ob eine Headerdatei bereits inkludiert wurde, damit sie nicht ein zweites Mal inkludiert wird. Wenn Sie also eine Headerdatei erstellt haben, sollten Sie immer Folgendes einfügen, damit die Datei nicht mehrmals inkludiert wird:

```
// myheader.h
#ifndef MYHEADER_H_
#define MYHEADER_H_

 // Der eigentliche Quellcode für die Headerdatei myheader.h

#endif
```

Der Präprozessor überprüft, ob er die Headerdatei `myheader.h` bereits eingebunden hat. Beim ersten Inkludieren ist das Makro `MYHEADER_H_` noch nicht definiert, sodass der Inhalt zwischen `#ifndef` und `#endif` in die Quelldatei eingefügt wird. Wird jetzt erneut `myheader.h` von einer anderen Datei inkludiert, ist `#ifndef` falsch. Alles zwischen `#ifndef` und `#endif` wird jetzt vom Präprozessor ignoriert.

Probleme beim mehrfachen Inkludieren

Mehrfaches Inkludieren sollte vermieden werden, wenn Sie mehrere Headerdateien und Module verwenden, weil dies zu diversen Compilerfehlern führen kann.

In der Tabelle 8.2 finden Sie einen Überblick über die restlichen Präprozessor-Direktiven.

Direktive	Beschreibung
`#error "zeichenkette"`	Das Programm lässt sich nicht übersetzen und gibt die Fehlermeldung `zeichenkette` zurück. Damit können Sie verhindern, dass ein nicht fertiggestellter oder nicht fehlerfreier Code einfach so übersetzt wird. Das ist recht praktisch als Hinweis, wenn mehrere Personen an einem Programm arbeiten.
`#line n` `#line n "dateiname"`	Diese Direktive hat keinen Einfluss auf das Programm selbst. Damit können Sie beispielsweise die Zeilennummer in einem Programm mit `n` festlegen und den Dateinamen auf `dateinamen` setzen. Die Nummerierung hat dann Einfluss auf die vordefinierten Präprozessor-Direktiven `__LINE__` und `__FILE__`.

Tabelle 8.2 Die restlichen Präprozessor-Direktiven

Direktive	Beschreibung
`#pragma`	`#pragma`-Direktiven sind compilerspezifisch, also von Compiler zu Compiler verschieden. Wenn ein Compiler eine bestimmte `#pragma`-Direktive nicht kennt, wird diese ignoriert. Mithilfe der Pragmas können beispielsweise Compileroptionen definiert werden, ohne mit anderen Compilern in Konflikt zu geraten.
`#`	Ein leeres `#` wird vom Präprozessor entfernt und hat keine Auswirkung im Programm.

Tabelle 8.2 Die restlichen Präprozessor-Direktiven (Forts.)

_Pragma-Operator

Damit Sie mit `pragma` auch Makros verwenden können, wurde im C99-Standard der Operator `_Pragma( bezeichner)` eingeführt. Mit `#pragma` war dies ja nicht möglich, weil es sich schon um eine Direktive handelte.

Des Weiteren gibt es noch einige vordefinierte Makros, die u. a. für das Debuggen von Programmen recht nützlich sein können.

Makroname	Beschreibung
`__LINE__`	Gibt eine Ganzzahl der aktuellen Zeilennummer in der Programmdatei zurück.
`__FILE__`	Gibt den Namen der Programmdatei als String-Literal zurück.
`__DATE__`	Gibt das Übersetzungsdatum der Programmdatei als String-Literal zurück.

Tabelle 8.3 Vordefinierte Standardmakros

Makroname	Beschreibung
`__TIME__`	Gibt die Übersetzungszeit der Programmdatei als String-Literal zurück.
`__STDC__`	Besitzt diese ganzzahlige Konstante den Wert 1, wurde das Programm mit dem Standard kompiliert.
`__func__`	Gibt den Namen der Funktion aus, in der das Makro verwendet wird (seit C99).
`__STD_VERSION__`	Enthält eine ganzzahlige Konstante vom Typ `long`, welche den Standard beschreibt, mit dem das Programm kompiliert wurde. Der Wert 199409L steht für C90, 199901L für C99 und 201112L für C11.

Tabelle 8.3 Vordefinierte Standardmakros (Forts.)

Hier ein Beispiel, das einige dieser vordefinierten Makros im Einsatz zeigt:

```
00 // kap008/listing006.c
01 #include <stdio.h>
02 #include <stdlib.h>

03 #if __STDC_VERSION__ >= 199901L
04 void eineFunktion(void) {
05   printf("Name der Funktion: %s\n", __func__);
06 }
07 #else
08 void eineFunktion(void) {
09   printf("Compiler kann kein C99\n");
10 }
11 #endif
```

```
12 int main(void) {
13 #if __STDC_VERSION__ >= 201112L
14   printf("Die Ausrichtung von char: %zd\n",_Alignof(char));
15 #else
16   printf("Compiler kann kein C11\n");
17 #endif

18   printf("Datum der Uebersetzung: %s\n", __DATE__);
19   printf("Zeit der Uebersetzung : %s\n", __TIME__);
20   printf("Zeile: %3d | Datei: %s\n", __LINE__, __FILE__);
21   eineFunktion();
22   return EXIT_SUCCESS;
23 }
```

Das Formatelement `%s` wurde im Buch noch nicht behandelt. Sie verwenden es in `printf`, wenn eine Stringvariable als Parameter übergeben wird. Das Umwandlungszeichen `s` des Formatstrings steht für String. Im Listing geben beispielsweise die vordefinierten Makros wie `__DATE__`, `__TIME__`, `__FILE__` eine Stringvariable mit entsprechenden Inhalt zurück. Ansonsten spricht das Listing für sich selbst.

Das Programm bei der Ausführung:

```
Die Ausrichtung von char: 1
Datum der Uebersetzung: Nov 25 2015
Zeit der Uebersetzung : 17:06:40
Zeile:  24 | Datei: Projects/listing008/listing006.c
Name der Funktion: eineFunktion
```

8.4 Programmdiagnose mit assert()

Die Definition des Makros `assert` (und auch `static_assert`; siehe Abschnitt 3.9.4, »Sicherheit beim Kompilieren mit _Static_assert«) in der Headerdatei `<assert.h>` hängt vom Makro `NDEBUG` ab, welches nicht von `<assert.h>` definiert wird. Wenn Sie `NDEBUG` definieren, werden alle im Code folgenden `assert`-Makros vom Compiler ignoriert. Wichtig ist es dabei

auch, dass Sie NDEBUG vor dem Inkludieren von assert definieren. Definieren Sie NDEBUG nicht, können Sie das Makro assert verwenden, um Ausdrücke zur Laufzeit auf logische Fehler zu testen. Ist der Ausdruck gleich 0, wird eine Fehlermeldung als Diagnose ausgegeben und das Programm mit abort() abgebrochen. Gerade wenn Sie anfangen, das Programm zu entwickeln und zu testen, sind solche *Assertionen* sehr hilfreich.

Hier ein einfaches Beispiel, welches das assert-Makro in der Praxis demonstriert:

```
00  // kap008/listing007.c
01  #include <stdio.h>
02  #include <stdlib.h>
03  #include <math.h>
04  // #define NDEBUG
05  #include <assert.h>

06  int main(void) {
07    double x = 0.0;
08    printf("Bitte eine positive Gleitkommazahl: ");
09    if( scanf("%lf", &x) != 1 ) {
10      printf("Fehler bei der Eingabe ...\n");
11      return EXIT_FAILURE;
12    }
13    assert(x >= 0.0);
14    double dval = sqrt(x);
15    printf("%lf\n", dval);
16    return EXIT_SUCCESS;
17  }
```

In diesem Beispiel wird in der Zeile **(13)** überprüft, ob die Gleitkommazahl größer bzw. gleich 0.0 ist. Solange dieser Gesamtausdruck nicht ungleich 0 ist, wird das Programm ordnungsgemäß ausgeführt. Das Programm bei der Ausführung:

```
Bitte eine positive Gleitkommazahl: 3.1
1.760682
```

```
Bitte eine positive Gleitkommazahl: -3.3
Assertion failed: file /Projects/listing007/
listing007.c, func main, line 15, x >= 0.0
abort -- terminating
*** Process returned 1 ***
```

Das Programm bei der Ausführung, wobei das auskommentierte NDEBUG in der Zeile **(04)** definiert wurde:

```
Bitte eine positive Gleitkommazahl: -3.3
nan
```

assert() für die Entwicklungsphase

Der Einsatz von assert() sollte der Entwicklungsphase vorbehalten sein. Ein Anwender kann in der Regel nicht viel mit der Fehlerausgabe anfangen. Wenn Sie das Programm freigeben, sollten Sie für das Release NDEBUG definieren, wodurch jedes assert im Programm durch ein Null-Makro ersetzt wird. Erwägen Sie auch die Verwendung von static_assert, womit Sie eine Überprüfung bereits während der Kompilierzeit vornehmen können. static_assert ist seit C11 dabei und wurde im Buch bereits im Abschnitt 3.9.4, »Sicherheit beim Kompilieren mit _Static_assert«, behandelt.

8.5 Generische Auswahl

In C++ ist es möglich, Funktionen zu überladen, wobei die Funktionen mit verschiedenen Parameterlisten denselben Bezeichner haben dürfen. Anhand des Datentyps des Übergabeparameters erkennt der Compiler dann die entsprechende Funktion. Dies kann praktisch sein, weil so eine Funktion immer mit dem gleichen Namen, aber unterschiedlichem Datentyp aufgerufen und verwendet werden kann. Ein einfaches Beispiel hierzu:

```
void genfunc_integer(int i) { … }
void genfunc_double(double d) { … }
void genfunc_complex(double complex c) { … }
```

Ein echtes Überladen kann zwar mit C nicht realisiert werden, aber seit C11 wurde die generische Auswahl (engl. *generic selection*) eingeführt, die eine ähnliche Funktionalität bietet und die unterschiedlich bezeichnete Funktion anhand des übergebenen Datentyps auswählt. Hierzu wurde das Schlüsselwort `_Generic` eingeführt. Mit diesem Schlüsselwort können Sie Makros definieren, mit denen Sie eine Fallunterscheidung aufgrund des Datentyps treffen können.

Hier ein einfaches Anwendungsbeispiel, das die Verwendung von `_Generic` in der Praxis zeigt:

```
00 // kap008/listing008.c
01 #include <stdio.h>
02 #include <complex.h>
03 #include <stdlib.h>

04 #define genfunc(X) _Generic ((X),  \
     default: genfunc_integer,       \
     double: genfunc_double,         \
     double complex: genfunc_complex \
   )(X)

05 void genfunc_integer(int i) {
06   printf("Integer: %d\n", i);
07 }

08 void genfunc_double(double d) {
09   printf("Double: %lf\n", d);
10 }

11 void genfunc_complex(double complex c) {
12   printf("Complex: %lf %lf\n", creal(c), cimag(c));
13 }

14 int main(void) {
15   int ival = 12345;
16   double dval = 3.14;
17   double complex dc = 2.2 + 3.3 * I;
```

```
18    genfunc(ival); // verwendet genfunc_integer
19    genfunc(dval); // verwendet genfunc_double
20    genfunc(dc);   // verwendet genfunc_complex
21    return EXIT_SUCCESS;
22  }
```

In der Zeile **(04)** können Sie die Definition des Makros `genfunc(X)` sehen, wobei X der Platzhalter für den zu übergebenden Datentyp ist. Nach dem Schlüsselwort `_Generic` stehen die Zuweisungen des entsprechenden Datentyps und der entsprechenden Funktion. Die Funktionen selbst werden in den Zeilen **(05)** bis **(13)** definiert. In der `main`-Funktion in den Zeilen **(18)**, **(19)** und **(20)** sehen Sie auch den Vorteil von `_Generic`. Hierbei brauchen Sie nur noch `genfunc()` verwenden, und es wird automatisch die passende Funktion aufgerufen, wenn für den übergebenen Datentyp eine existiert. Passt kein Typ, wird die Funktion aufgerufen, die hinter `default` notiert wurde (im Beispiel die Version für den Typen `int`).

Das Beispiel bei der Ausführung:

```
Integer: 12345
Double: 3.140000
Complex: 2.200000 3.300000
```

8.6 Eigene Header erstellen

Wenn die Programme umfangreicher werden, sollten Sie den Quellcode in verschiedene sinnvolle Module zerlegen. In der Praxis teilt man hierbei häufig ein Projekt in eine Quell- und eine Headerdatei, wodurch diese Module in mehreren Programmen verwenden werden können. In einem Header werden gewöhnlich die Funktionsdeklarationen (Prototypen), globale Variablendeklaration (wenn unbedingt nötig), Konstanten und eigene Typdefinitionen notiert.

Eine einfache Headerdatei könnte demnach wie folgt aussehen:

```
00  // kap008/modulA.h
01  #ifndef MODULA_H
02  #define MODULA_H
```

```
03  #define PI 3.14159265359
04  extern int modulA_var;
05  double kreisflaeche(double);

06  #endif /* MODULA_H */
```

Mit den Zeilen **(01)** prüfen Sie, ob dem Präprozessor bereits die Direktive `MODULA_H` bekannt ist. Ist dies nicht der Fall, definieren Sie diese Direktive in der Zeile **(02)**, um so ein mehrfaches Inkludieren zu vermeiden. Dieser Name ist frei wählbar, muss aber im Projekt eindeutig sein. Ansonsten finden Sie hier noch die Konstante `PI`, und es werden eine globale Variable und ein Funktionsprototyp deklariert.

Wenn Sie eine Headerdatei geschrieben haben, wird in der Regel auch eine entsprechende C-Quelldatei dazu erstellt, die den Header inkludiert und unter anderem auch die Definitionen der Funktionsprototypen enthält. Im einfachen Beispiel sieht die C-Quelldatei wie folgt aus:

```
00  // kap008/modulA.c
01  #include "modulA.h"
02  #include <stdio.h>

03  int modulA_var = 1234;

04  double kreisflaeche(double r) {
05    return PI * r * r;
06  }
```

In der Praxis wird diese C-Quelldatei häufig gar nicht mit ausgeliefert und lediglich zusammen mit der Headerdatei kompiliert (ohne den Linkerlauf). Auf diese Weise wird eine Objektdatei erzeugt, häufig in der Form *modulA.obj* oder *modulA.o*. Um die Funktionalitäten der Module in einem Programm zu verwenden, reicht es daher aus, wenn die Headerdatei und die Objektdatei ausgeliefert und verwendet werden. Damit ersparen Sie es sich, jedes Mal das komplette Projekt mit allen einzelnen Dateien zu kompilieren.

Um nun auch die Funktionalitäten der Headerdatei in der Praxis zu testen, müssen Sie sie nur an der entsprechenden Stelle im Projekt einfügen und die Funktionen verwenden. Hierzu noch ein Beispiel, welches das kleine Beispielmodul in der Praxis nutzt:

```
00 // kap008/listing009.c
01 #include <stdio.h>
02 #include <stdlib.h>
03 #include "modulA.h"

04 int main(void) {
05   double dval = kreisflaeche(5.5);
06   printf("Kreisflaeche        : %lf\n", dval);
07   printf("Wert von modulA_var : %d\n", modulA_var);
08   return EXIT_SUCCESS;
09 }
```

Beachten Sie beim Übersetzen von mehreren Modulen, dass Sie auch die entsprechenden C-Quelldateien mit angeben bzw. zum Projekt hinzugefügt haben.

8.7 Kontrollfragen und Aufgaben

1. Erklären Sie den Unterschied zwischen dem Einbinden einer Datei mit `#include` mit eingeschlossenen Anführungszeichen (beispielsweise `"datei.h"`) und mit spitzen Klammern (beispielsweise `<datei.h>`).
2. Was kann mit der `define`-Direktive gemacht werden?
3. Wie können Sie eine symbolische Konstante oder ein Makro aufheben?
4. Beachten Sie folgende symbolische Konstante:

   ```
   #include <math.h>   // benötigter Header für sqrt()
   #define VAL sqrt(2)*8
   ```

   Und folgende Konstante mit dem Schlüsselwort `const`:

   ```
   #include <math.h>   // benötigter Header für sqrt()
   const double VAL = sqrt(2)*8;
   ```

Welche der beiden Konstanten wäre die bessere Alternative im Programm und warum?

5. Was verstehen Sie unter einer bedingten Kompilierung?
6. Im folgenden Beispiel gibt die Multiplikation den Wert 190 zurück. Korrekt wäre allerdings der Wert 100 (10*(20-10)). Wie können Sie das Problem beheben?

```
00 // kap008/aufgabe001.c
01 #include <stdio.h>
02 #include <stdlib.h>
03 #define MULTI(a, b) (a*b)

04 int main(void) {
05   int val1 = 10, val2 = 20;
06   printf("Multiplikation = %d\n", MULTI(val1, val2-10));
07   return EXIT_SUCCESS;
08 }
```

7. Wie oft wird die `for`-Schleife ausgeführt und warum?

```
00 // kap008/aufgabe002.c
01 #include <stdio.h>
02 #include <stdlib.h>
03 #define CNT 10

04 int main(void) {
05   int i;
06 #undef CNT
07 #define CNT 5
08   for( i = 0; i < CNT; i++) {
09 #undef CNT
10 #define CNT 20
11     printf("%d\n", i);
12   }
13   return EXIT_SUCCESS;
14 }
```

8. Entwickeln Sie Ihre eigene kleine Makrosprache. Bringen Sie mithilfe der `define`-Direktive dieses kleine Hauptprogramm (`Hallo Welt`) in C zur

Ausführung. Erstellen Sie hierzu die Headerdatei `"mysyntax.h"`, und berücksichtigen Sie auch, dass diese Headerdatei nicht mehrfach inkludiert werden kann.

```
00 // kap008/aufgabe003.c
01 #include "mysyntax.h"

02 MAIN
03 OPEN
04 WRITE"Hallo Welt\n"WRITE_
05 END
06 CLOSE
```

9. Definieren Sie zwei parametrisierte Makros, die von zwei Werten den maximalen und den minimalen Wert zurückgeben.
10. Schreiben Sie ein Makro, das mit einem symbolischen Bezeichner wie `DEBUG_ALL` Informationen wie Datum, Uhrzeit, Zeile und Datei zu Debugging-Zwecken ausgibt.

Kapitel 9

Arrays und Zeichenketten (Strings)

Bisher haben Sie sich auf einfache Datentypen beschränkt. Bei den Aufgaben wurden lediglich ganze Zahlen (`char`, `short`, `int`, `long`, `long long`) bzw. Fließkommazahlen (`float`, `double`, `long double`) besprochen. In diesem Kapitel erfahren Sie nun etwas über zusammengesetzte Daten, kurz *Arrays*. Wenn der Typ des Arrays beispielsweise `T` ist, spricht man von einem `T`-Array. Ist das Array vom Typ `long`, spricht man von einem `long`-Array.

9.1 Arrays verwenden

Mit Arrays werden die einzelnen Elemente als Folge von Werten eines bestimmten Typs abgespeichert und bearbeitet. Arrays werden auch als Vektoren, Felder oder Reihungen bezeichnet.

9.1.1 Arrays definieren

Die allgemeine Syntax zur Definition eines Arrays sieht wie folgt aus:

```
Datentyp Arrayname[Anzahl_der_Elemente];
```

Als `Datentyp` geben Sie an, von welchem Typ die Elemente des Arrays sein sollen. Der `Arrayname` ist frei wählbar, mit denselben Einschränkungen für Bezeichner wie bei Variablen (siehe Abschnitt 2.4.1, »Bezeichner«). Mit `Anzahl_der_Elemente` wird die Anzahl der Elemente angegeben, die im Array gespeichert werden kann. Der Wert in den eckigen Klammern muss eine ganzzahlige Konstante oder ein ganzzahliger kontanter Ausdruck und größer als 0 sein. Ein Array, das aus Elementen unterschiedlicher Datentypen besteht, gibt es in C nicht.

Array mit variabler Länge (VLA)

Im C11-Standard sind Variablen für die Anzahl der Elemente in den eckigen Klammern nur als optionale Erweiterung erlaubt, werden aber nicht mehr gefordert, wie dies noch in C99 der Fall war.

Zugreifen können Sie auf das gesamte Array mit allen Komponenten über den Arraynamen. Die einzelnen Elemente eines Arrays werden durch den Arraynamen und einen Indexwert zwischen eckigen Klammern `[n]` angesprochen. Der Indexwert selbst wird über eine Ordinalzahl (Ganzzahl) angegeben und fängt bei **0** an zu zählen.

Nehmen wir als Beispiel folgendes Array:

```
int iArray[8];
```

Das Array hat den Bezeichner `iArray` und besteht aus acht Elementen vom Typ `int`. In diesem Array können Sie also acht Integerwerte abspeichern. Intern wird für dieses Array somit automatisch Speicherplatz für acht Arrays vom Typ `int` reserviert. Wie viel Speicher dies ist, können Sie mit `8*sizeof(int)` ermitteln. Hat der Typ `int` auf Ihrem System eine Breite von 4 Bytes, ergibt dies 32 Bytes (8 Elemente × 4 Bytes pro Element = 32).

9.1.2 Arrays mit Werten versehen und darauf zugreifen

Um einzelnen Arrayelementen einen Wert zu übergeben oder Werte daraus zu lesen, wird der Indizierungsoperator `[]` (auch Subskript-Operator genannt) verwendet. Wie der Name schon sagt, können Sie damit auf ein Arrayelement mit einem Index zugreifen.

Hierzu ein einfaches Beispiel:

```
00  // kap009/listing001.c
01  #include <stdio.h>
02  #include <stdlib.h>

03  int main(void) {
04    int iArray[3] = {0};
05    // Array mit Werten initialisieren
```

```
06    iArray[0] = 1234;
07    iArray[1] = 3456;
08    iArray[2] = 7890;

09    // Inhalt der Arrayelemente ausgeben
10    printf("iArray[0] = %d\n", iArray[0]);
11    printf("iArray[1] = %d\n", iArray[1]);
12    printf("iArray[2] = %d\n", iArray[2]);
13    return EXIT_SUCCESS;
14 }
```

In den Zeilen **(06)** bis **(08)** wurden je drei Werte mithilfe des Indizierungsoperators und der entsprechenden Indexnummer jeweils einem Wert zugewiesen. Gleiches wurde in den Zeilen **(10)** bis **(12)** gemacht, mit dem Unterschied, dass hier die Werte ausgegeben wurden.

Ihnen dürfte gleich auffallen, dass in Zeile **(04)** ein Array mit der Ganzzahl 3 zwischen dem Indizierungsoperator definiert wurde, aber weder bei der Zuweisung in den Zeilen **(06)** bis **(08)** noch bei der Ausgabe in den Zeilen **(10)** bis **(12)** wurde vom Indexwert 3 Gebrauch gemacht. Darüber stolpern viele Einsteiger: Das erste Element in einem Array muss nämlich immer die Indexnummer 0 sein. Wenn das erste Element in einem Array den Index 0 hat, besitzt das letzte Element logischerweise den Wert `n-1` (`n` ist die Arraygröße).

Unser Array `iArray` mit drei Elementen vom Datentyp `int` aus dem Beispiel *listing001.c* können Sie sich wie in Abbildung 9.1 vorstellen.

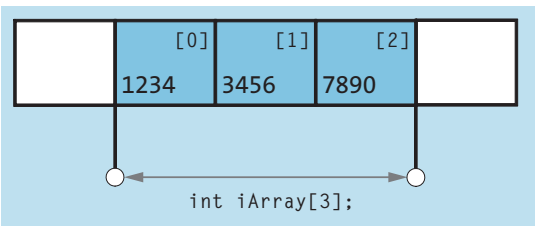

Abbildung 9.1 Dies ist ein Array mit drei Elementen, das mit Werten initialisiert wurde.

Hätten Sie im Programm *listing001.c* folgende Zeile hinzugefügt:

```
...
iArray[3] = 6666;
...
printf("iArray[3] = %d\n", iArray[3]);
```

würden Sie auf einen nicht geschützten und reservierten Speicherbereich zugreifen. Bestenfalls stürzt das Programm gleich mit einer *Schutzverletzung* (*segmentation fault*) oder *Zugriffsverletzung* (*access violation*) ab. Schlimmer ist es aber, wenn das Programm weiterläuft und irgendwann eine andere Variable diesen Speicherbereich, der ja nicht reserviert und geschützt ist, verwendet und ändert. Sie erhalten dann unerwünschte Ergebnisse bis hin zu einem schwer auffindbaren Fehler im Programm. In Abbildung 9.2 sehen Sie eine grafische Darstellung der Schutzverletzung des Speichers.

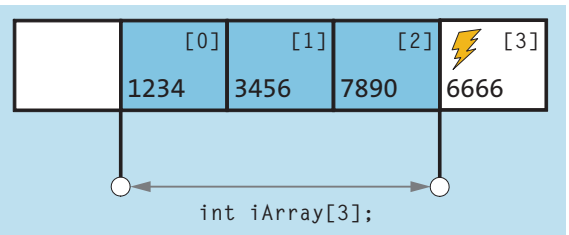

Abbildung 9.2 Mithilfe des Indizierungsoperators wurde auf einen nicht geschützten Bereich zugegriffen, was eine Schutzverletzung darstellt. Das weitere Verhalten des Programms ist undefiniert.

Array-Überlauf überprüfen

Auf vielen Systemen gibt es eine Compiler- bzw. Debugging-Option, mit der eine Schutzverletzung eines Arrays geprüft und mindestens eine Warnmeldung ausgegeben wird, wenn ein möglicher Zugriff auf einen nicht geschützten Bereich erkannt wird.

Beispiele wie das *listing001.c* sind ziemlich trivial. Häufig werden Sie Werte von Arrays in Schleifen übergeben oder auslesen. Hierbei kann es

schnell mal zu einen Über- bzw. Unterlauf kommen, wenn Sie nicht aufpassen. Ein einfaches und typisches Beispiel dazu:

```
00 // kap009/listing002.c
01 #include <stdio.h>
02 #include <stdlib.h>
03 #define MAX 10

04 int main(void) {
05   unsigned int iArray[MAX];
06   // Werte an alle Elemente
07   for(unsigned int i = 0; i < MAX; i++) {
08     iArray[i]=i*i;
09   }
10   // Werte ausgeben
11   for(unsigned int i = 0; i < MAX; i++) {
12     printf("iArray[%d] = %d\n", i, iArray[i]);
13   }
14   return EXIT_SUCCESS;
15 }
```

Im Programm wird nichts anderes gemacht, als dem Array `iArray` mit `MAX`-Elementen vom Typ `int` in der `for`-Schleife (Zeile **(07)** bis **(09)**) Werte einer Multiplikation zuzuweisen. Diese Werte werden in den Zeilen **(11)** bis **(13)** auf ähnlichem Weg wieder ausgegeben.

Das Programm bei der Ausführung:

```
iArray[0] = 0
iArray[1] = 1
iArray[2] = 4
iArray[3] = 9
iArray[4] = 16
iArray[5] = 25
iArray[6] = 36
iArray[7] = 49
iArray[8] = 64
iArray[9] = 81
```

In einer solchen `for`-Schleife sollten Sie immer darauf achten, dass es nicht zu einem Über- bzw. Unterlauf des Arrays kommt. Vergisst man hier, dass das erste Element eines Arrays mit dem Index 0 beginnt, macht man den folgenden fatalen Fehler:

```
unsigned iArray[10];
...
for(unsigned int i = 0; I <= 10; i++) {
   iArray[i]=i;
}
```

Durch die Verwendung des <=-Operators statt des <-Operators werden jetzt 11 anstelle von 10 Arrays initialisiert. Damit hätten Sie einen Array-Überlauf erzeugt. Gleiches gilt bei einem Unterlauf eines Arrays, wenn Sie den Array-Index beispielsweise rückwärts durchlaufen. Auch hier müssen Sie dafür sorgen, dass kein negativer Indexwert für ein Array verwendet wird.

Initialisierung mit einer Initialisierungsliste

Ein Array können Sie bereits bei der Definition mit einer Initialisierungsliste initialisieren. Hierbei wird bei der Definition eine Liste von Werten in geschweiften Klammern, getrennt durch Kommata, an das Array zugewiesen. Ein einfaches Beispiel:

```
float fArray[3] = { 0.75, 1.0, 0.5 };
```

Nach dieser Initialisierung haben die einzelnen Elemente im Array `fArray` folgende Werte:

```
fArray[0] = 0.75
fArray[1] = 1.0
fArray[2] = 0.5
```

In der Definition eines Arrays mit Initialisierungsliste kann auch die Längenangabe fehlen. So ist die folgende Definition gleichwertig mit der obigen:

```
// float-Array mit drei Elementen
float fArray[] = { 0.75, 1.0, 0.5 };
```

Geben Sie hingegen bei der Längenangabe einen größeren Wert an als Elemente in der Initialisierungsliste vorhanden sind, haben die restlichen Elemente in der Liste automatisch den Wert 0. Wenn Sie mehr Elemente angeben als in der Längenangabe definiert, werden die zu viel angegebenen Werte in der Initialisierungsliste einfach ignoriert. Hier ein Beispiel:

```
long lArray[5] = { 123, 456 };
```

Es wurden nur die ersten beiden Elemente in der Liste initialisiert. Die restlichen drei Elemente haben automatisch den Wert 0. Nach der Initialisierung haben die einzelnen Elemente im Array `lArray` folgende Werte:

```
lArray[0] = 123
lArray[1] = 456
lArray[2] = 0
lArray[3] = 0
lArray[4] = 0
```

Somit können Sie davon ausgehen, dass Sie die Werte der einzelnen Elemente von einem lokalen Array mit der Definition 0 initialisieren können:

```
// Alle 100 Arrayelemente mit 0 initialisiert
int iarray[100] = { 0 };
```

Ohne explizite Angabe einer Initialisierungsliste werden die einzelnen Elemente nur bei globalen oder `static`-Arrays automatisch vom Compiler mit 0 initialisiert:

```
// Alle 100 Arrayelemente automatisch mit 0 initialisiert
static int iarray[100];
```

Bestimmte Elemente direkt initialisieren

Mit dem C99-Standard wurde auch die Möglichkeit eingeführt, ein bestimmtes Element bei der Definition zu initialisieren. Hierzu müssen Sie in der Initialisierungsliste lediglich das gewünschte Element in eckigen Klammern angeben. Hier ein Beispiel:

```
#define MAX 5
int iArray[MAX] = { 123, 456, [MAX-1] = 789 };
```

Es wurden die ersten beiden Elemente initialisiert, anschließend wurde dem letzten Wert in der Liste ein Wert zugewiesen. Nach der Initialisierung haben die einzelnen Elemente im Array `iArray` folgende Werte:

```
iArray[0] = 123
iArray[1] = 456
iArray[2] = 0
iArray[3] = 0
iArray[4] = 789
```

Array mit Schreibschutz

Wenn Sie ein Array benötigen, bei dem die Werte schreibgeschützt sind und nicht mehr verändert werden sollen, können Sie das Schlüsselwort `const` vor die Array-Definition setzen. Die Werte in der Initialisierungsliste können so nicht mehr aus Versehen geändert und überschrieben werden. Ein einfaches Beispiel dazu:

```
#define RGB 3
// Konstantes Array kann zur Laufzeit nicht geändert werden.
const unsigned int gelb[RGB] = { 255, 255, 0 };
// Fehler!!! Zugriff auf konstantes Array nicht möglich
gelb[2] = 20;
```

Arrays mit fester und variabler Länge (VLA) (optional seit C11)

Mit dem C99-Standard wurde ebenfalls eingeführt, dass bei der Definition die Anzahl der Elemente kein konstanter Ausdruck mehr sein muss. Seit dem C11-Standard ist die VLA-Unterstützung (VLA = *variable length arrays*) allerdings nur noch optional vorgeschrieben. Trotzdem soll sie hier kurz beschrieben werden. Mit dem Makro `__STDC_NO_VLA__` können Sie testen, ob VLA unterstützt wird.

Voraussetzung dafür, dass die Anzahl der Elemente kein konstanter Ausdruck sein muss, ist, dass das Array eine lokale Variable ist und nicht mit dem Spezifizierer `static` gekennzeichnet ist. Das Array muss außerdem in einem Anweisungsblock definiert werden. Hierzu ein Beispiel:

```
00  // kap009/listing003.c
01  #include <stdio.h>
02  #include <stdlib.h>
```

```
03 #if __STDC_NO_VLA__
04   #error "No VLA support!"
05 #endif

06 int main(void) {
07   int val = 0;
08   printf("Anzahl der Elemente: ");
09   if( scanf("%d", &val) != 1 ) {
10     printf("Fehler bei der Eingabe ...\n");
11     return EXIT_FAILURE;
12   }
13   if(val > 0) {
14     int iarr[val];
15     for(int i = 0; i < val; i++) {
16       iarr[i] = i;
17     }
18     for(int i = 0; i < val; i++) {
19       printf("%d\n", iarr[i]);
20     }
21   }
22   return EXIT_SUCCESS;
23 }
```

In Zeile **(14)** sehen Sie die Definition eines `int`-Arrays, dessen Elementanzahl beim Start des Programms noch nicht bekannt ist und erst vom Anwender bestimmt wird. Dass dies tatsächlich funktioniert, können Sie in Zeile **(16)** erkennen. Dort wurde den einzelnen Elementen ein Wert zugewiesen. In Zeile **(19)** werden die Werte der einzelnen Elemente ausgegeben. Damit dies im Beispiel überhaupt funktioniert, ist es wichtig, dass die Definition in Zeile **(14)** in einem Anweisungsblock zwischen den Zeilen **(13)** bis **(21)** steht. Nur innerhalb dieses Bereichs ist das VLA-Array `iarr` gültig.

In der Praxis spricht somit nichts dagegen, die variable Länge von Arrays auch in Funktionen zu verwenden. Hier ein Beispiel:

```
void varArray( int v ) {
  float fvarr[v];
  ...
```

```
}
...
// Funktionsaufruf
varArray(25);
```

9.1.3 Arrays mit scanf einlesen

Das Einlesen von einzelnen Array-Werten funktioniert im Grunde genommen genauso wie mit gewöhnlichen Variablen. Sie haben allerdings neben dem Adressoperator noch den Indizierungsoperator `[]`. Zwischen den eckigen Klammern geben Sie den Wert für den Index an, mit dem Sie das eingelesene Element versehen wollen. Ein Beispiel dazu:

```
00 // kap009/listing004.c
01 #include <stdio.h>
02 #include <stdlib.h>
03 #define MAX 3

04 int main(void) {
05   double dval[MAX];
06   for(int i = 0; i < MAX; i++) {
07     printf("%d. double-Wert: ", i+1);
08     if (scanf("%lf", &dval[i]) != 1) {
09       printf("Fehler bei der Eingabe ...\n");
10       return EXIT_FAILURE;
11     }
12   }
13   printf("Sie gaben ein: ");
14   for(int i = 0; i < MAX; i++) {
15     printf("%.2lf ", dval[i]);
16   }
17   printf("\n");
18   return EXIT_SUCCESS;
19 }
```

Abgesehen von Zeile **(08)**, in der mithilfe des Adressoperators, des Indizierungsoperators und des entsprechenden Indexwertes `MAX` Werte in das

Array eingelesen werden, enthält das Listing nichts Unbekanntes. Das Programm bei der Ausführung:

```
1. double-Wert: 3.1
2. double-Wert: 3
3. double-Wert: 0.55
Sie gaben ein: 3.10 3.00 0.55
```

9.1.4 Arrays an Funktionen übergeben

An dieser Stelle komme ich nicht umhin, auf Abschnitt 10.4 vorzugreifen, weil Arrays an Funktionen nicht wie andere Variablen als Kopie übergeben werden können, sondern als Adresse übergeben werden müssen. Somit übergeben Sie hier kein komplettes Element bzw. das komplette Array als Kopie an die Funktion, sondern nur noch eine (Anfangs-)Adresse auf dieses Array und dessen Länge in einem weiteren Parameter.

Deshalb wirken sich Änderungen an diesen Werten auch auf den Aufrufer aus. Sie greifen dann direkt auf die Adressen der einzelnen Arrayelemente des Aufrufers zu.

Arrays werden sequenziell gespeichert

Wenn Sie die Anfangsadresse von einem Array an eine Funktion übergeben, können Sie sich darauf verlassen, dass die einzelnen Arrayelemente im Speicher sequenziell abgelegt sind bzw. sein müssen. Deshalb genügt es, die Anfangsadresse und die Länge eines Arrays an eine Funktion zu übergeben, um auf das gesamte Array zugreifen zu können.

Sie übergeben ein Array an eine Funktion, indem Sie außerdem auch einen zusätzlichen formalen Parameter erstellen. Dort können Sie die Anzahl der Elemente des Arrays mit an die Funktion übergeben.

Hierzu ein einfaches Beispiel:

```
00  // kap009/listing005.c
01  #include <stdio.h>
02  #include <stdlib.h>
```

```
03  #define MAX 3

04  void readArray( int arr[], int n ) {
05    for(int i=0; i < n; i++) {
06      printf("[%d] = %d\n", i, arr[i]);
07    }
08  }

09  void initArray( int arr[], int n) {
10    for(int i=0; i < n; i++) {
11      arr[i] = i+i;
12    }
13  }

14  int main(void) {
15    int iArr[MAX];
16    initArray( iArr, MAX );
17    readArray( &iArr[0], MAX );
18    return EXIT_SUCCESS;
19  }
```

In Zeile **(16)** übergeben Sie die Adresse des in Zeile **(15)** definierten Arrays und die Anzahl der Elemente an die Funktion `initArray`. Diese ist in den Zeilen **(09)** bis **(13)** definiert. In der Funktion initialisieren Sie die einzelnen Elemente des Arrays mit Werten. In Zeile **(17)** des Programms übergeben Sie die Anfangsadresse des Arrays mit der Anzahl der Elemente an die Funktion `readArray` (Zeilen **(04)** bis **(08)**). Dort werden die einzelnen Elemente des Arrays ausgegeben. Beide Schreibweisen der Zeilen **(16)** und **(17)** sind übrigens gleichwertig; mit Zeile **(17)** übergeben Sie die Adresse des ersten Elements aber direkt an die Funktion.

Es wäre theoretisch möglich, die Adresse des zweiten Elements im Array mit

```
readArray(&iArr[1], MAX-1);
```

an die Funktion zu übergeben. Allerdings müssen Sie dann auch die Anzahl der Elemente entsprechend anpassen, um einen Überlauf zu vermeiden.

9.2 Mehrdimensionale Arrays

Arrays, wie sie bisher besprochen wurden, können Sie sich als einen sequenziellen Strang von hintereinander aufgereihten Werten vorstellen. In der Praxis spricht man dann von einem eindimensionalen Array. Es ist aber auch möglich, Arrays mit mehr als nur einer Dimension zu verwenden:

```
// Zweidimensionales Array mit 2 Zeilen und 3 Spalten
int tabelle[2][3];
```

Hier wurde z. B. ein zweidimensionales Array mit dem Namen `tabelle` definiert. Dies entspricht im Prinzip einem Array, dessen Elemente wieder Arrays sind. Die ersten Elemente `tabelle[0]` und `tabelle[1]` sind die Zeilen. Jede dieser Zeilen enthält ein weiteres Array mit drei `int`-Elementen. Somit besteht das Array `tabelle` aus sechs Elementen vom Typ `int`. Im Zusammenhang mit zweidimensionalen Arrays wird häufig auch von Matrizen gesprochen. Sie können sich dieses mehrdimensionale Array wie bei einer Tabellenkalkulation vorstellen (siehe Abbildung 9.3).

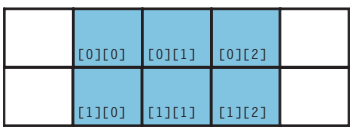

Abbildung 9.3 Ein zweidimensionales Array (2 Zeilen × 3 Spalten)

9.2.1 Zweidimensionalen Arrays Werte zuweisen und darauf zugreifen

Im Grunde funktioniert die Initialisierung von mehrdimensionalen Arrays wie die von eindimensionalen (siehe Abschnitt 9.1.2, »Arrays mit Werten versehen und darauf zugreifen«). Anstelle eines Indizierungsoperators [ ] müssen nur zwei Zuweisungsoperatoren verwendet werden, um auf die einzelnen Arrayelemente zuzugreifen. Trotzdem gibt es einige Besonderheiten, die Sie beachten sollten. Wir werden im Folgenden darauf noch eingehen.

Hierzu ein Beispiel:

```
00 // kap009/listing006.c
01 #include <stdio.h>
02 #include <stdlib.h>

03 int main(void) {
04   int mdarray[2][3];
05   // Array mit Werten initialisieren
06   mdarray[0][0] = 12;
07   mdarray[0][1] = 23;
08   mdarray[0][2] = 34;
09   mdarray[1][0] = 45;
10   mdarray[1][1] = 56;
11   mdarray[1][2] = 67;
12   // Inhalt ausgeben
13   for(int i=0; i < 2; i++) {
14     for(int j=0; j < 3; j++ ) {
15       printf("[%d][%d] = %d\n", i, j, mdarray[i][j]);
16     }
17   }
18   return EXIT_SUCCESS;
19 }
```

In den Zeilen **(06)** bis **(11)** wurden je sechs Werte mithilfe der Indizierungsoperatoren und der entsprechenden Indexnummern zugewiesen. Analog dazu werden die Werte in den Zeilen **(13)** bis **(17)** ausgegeben. Solche verschachtelten `for`-Schleifen sind typisch für mehrdimensionale Arrays. Mit der äußeren `for`-Schleife durchlaufen Sie die einzelnen Zeilen, mit der inneren die einzelnen Spalten dieser Zeile. Die eckigen Klammern bei der Formatanweisung von `printf()` in Zeile **(15)** sind eine reine Design-Entscheidung und haben keine Bedeutung.

Das Programm bei der Ausführung:

```
[0][0] = 12
[0][1] = 23
[0][2] = 34
[1][0] = 45
[1][1] = 56
[1][2] = 67
```

Initialisieren mit Initialisierungsliste

Ein mehrdimensionales Array können Sie bei der Definition mit einer Initialisierungsliste ähnlich wie bei den eindimensionalen Arrays explizit initialisieren. Hierbei wird dem Array bei der Definition eine Liste von Werten, getrennt mit einem Komma, in geschweiften Klammern zugewiesen. Einzelne Zeilen werden gewöhnlich zwischen weiteren geschweiften Klammern zusammengefasst. Ein einfaches Beispiel:

```
int mdarray[2][3] = { {12, 23, 34},
                      {45, 56, 67} };
```

Nach dieser Initialisierung haben die einzelnen Elemente im Array `mdarray` folgende Werte:

```
mdarray[0][0] = 12
mdarray[0][1] = 23
mdarray[0][2] = 34
mdarray[1][0] = 45
mdarray[1][1] = 56
mdarray[1][2] = 67
```

Auf die zusätzlichen geschweiften Klammern hätten Sie aber theoretisch auch verzichten können. Mit folgender Definition hätten Sie dasselbe erreicht:

```
int mdarray[2][3] = { 12, 23, 34, 45, 56, 67 };
```

Wenn Sie die Arrayelemente in einer Liste von Initialisierungswerten angeben, können Sie die erste Dimension auch weglassen. So ist also auch Folgendes möglich:

```
int mdarray[][3] = { {12, 23, 34},
                     {45, 56, 67} };
// oder auch:
int mdarray[][3] = { 12, 23, 34, 45, 56, 67 };
```

Die erste Dimension wird dann anhand der angegebenen Anzahl in der zweiten Dimension und der vorhandenen Initialisierungselemente berechnet.

Unvollständige mehrdimensionale Arrays deklarieren

Bei der Deklaration von mehrdimensionalen Arrays darf die Längenangabe der ersten Dimension auch fehlen. Allerdings muss sie an einer anderen Stelle im Programm definiert werden.

Elemente, die in einem mehrdimensionalen Array bei der Definition mit der Initialisierungsliste nicht ausdrücklich initialisiert wurden, erhalten automatisch den Wert 0. Hier ein Beispiel:

```
int mdarray[2][3] = { { 1 },
                      { 2, 3 } };
```

Nach dieser Initialisierung haben die einzelnen Elemente im Array `mdarray` folgende Werte:

```
mdarray[0][0] = 1
mdarray[0][1] = 0
mdarray[0][2] = 0
mdarray[1][0] = 2
mdarray[1][1] = 3
mdarray[1][2] = 0
```

9.2.2 Zweidimensionale Arrays an eine Funktion übergeben

Die Übergabe von zweidimensionalen Arrays an eine Funktion ist natürlich auch möglich. Allerdings ist es auch hier häufig ein wenig verwirrend, wie der Funktionskopf aussieht.

Sehen Sie sich hierzu folgende Funktion an – sie demonstriert die Übergabe von mehrdimensionalen Arrays an Funktionen:

```
00  // kap009/listing006-md.c
01  #include <stdio.h>
02  #include <stdlib.h>
03  #define WOCHE 2
04  #define TAGE  7

05  void durchschnitt( int arr[][TAGE] ) {
```

```
06    int durchs=0;
07    for( int i=0; i < WOCHE; i++) {
08      for( int j=0; j < TAGE; j++) {
09        durchs+=arr[i][j];
10      }
11    }
12    printf("Besucher in %d Tagen\n", WOCHE*TAGE);
13    printf("Gesamt      : %d\n", durchs);
14    printf("Tagesschnitt: %d\n", durchs /(WOCHE*TAGE));
15  }

16  int main(void) {
17    int besucher[WOCHE][TAGE];
18    printf("Besucherzahlen eingeben\n\n");
19    for(int i=0; i < WOCHE; i++) {
20      for(int j=0; j < TAGE; j++) {
21        printf("Woche %d, Tag %d: ", i+1, j+1);
22        if( scanf("%d", &besucher[i][j]) != 1 ) {
23          printf("Fehler bei der Eingabe\n");
24          return EXIT_FAILURE;
25        }
26      }
27    }
28    durchschnitt( besucher );
29    return EXIT_SUCCESS;
30  }
```

In diesem Beispiel wird eine Besucherstatistik erstellt. Dabei werden die Besucherzahlen für jeden Tag der letzten zwei Wochen in Zeile **(22)** an das mehrdimensionale Array `besucher` übergeben. Sie können hierbei auch gleich sehen, wie Sie mit `scanf` einzelne Werte in ein zweidimensionales Array einlesen können. Auch hier dürfen Sie den Adressoperator `&` nicht vergessen. In Zeile **(28)** wird dann die Anfangsadresse des mehrdimensionalen Arrays an die Funktion `durchschnitt()` übergeben.

In der Funktion selbst machen wir in den Zeilen **(05)** bis **(15)** nichts anderes mehr, als die Besucherzahlen der einzelnen Wochen und Tage in der ver-

schachtelten `for`-Schleife zu addieren und am Ende die gesamte Besucherzahl und den Tagesdurchschnitt zu berechnen.

Das Programm bei der Ausführung:

```
Woche 1, Tag 1: 123
Woche 1, Tag 2: 234
Woche 1, Tag 3: 246
Woche 1, Tag 4: 467
Woche 1, Tag 5: 147
Woche 1, Tag 6: 268
Woche 1, Tag 7: 345
Woche 2, Tag 1: 134
Woche 2, Tag 2: 234
Woche 2, Tag 3: 232
Woche 2, Tag 4: 126
Woche 2, Tag 5: 105
Woche 2, Tag 6: 101
Woche 2, Tag 7: 223
Besucher in 14 Tagen
Gesamt       : 2985
Tagesschnitt : 213
```

9.2.3 Noch mehr Dimensionen ...

Natürlich ist die Anzahl der möglichen Dimensionen nicht nur auf zwei beschränkt. Sie können durchaus auch drei, vier oder mehr Dimensionen definieren. Alles bisher Beschriebene lässt sich also in ähnlicher Weise bei einem drei- oder vierdimensionalen Array anwenden. Ein Beispiel einer solchen Definition könnte sein:

```
#define YEAR  20
#define MONTH 12
#define DAY   31
...
double abrechnung[YEAR][MONTH][DAY];
```

9.3 Strings (Zeichenketten)

Vielleicht haben Sie sich schon gefragt, was passiert, wenn Sie ein Array vom Typ `char` (oder auch `wchar_t`) verwenden. Sie werden es schon vermuten: Mit einer Folge von `char`-Zeichen können Sie einen kompletten Text, einen sogenannten String, speichern, verarbeiten und ausgeben. Auf Ihre Frage, wie Sie Text in C verarbeiten können, bekommen Sie in diesem Kapitel also die Antworten. In C gibt es keinen eigenen Datentyp für solche Strings und daher auch keine Operatoren, die Strings als Operanden verwenden können.

Für Arrays vom Typ `char` gelten nicht nur die Einschränkungen herkömmlicher Arrays, sondern es muss auch darauf geachtet werden, dass die zusammenhängende Folge von Zeichen mit dem Null-Zeichen `'\0'` (auch Stringende-Zeichen genannt) abgeschlossen wird. Genau genommen heißt dies, dass die Länge eines `char`-Arrays immer ein Zeichen größer sein muss als die Anzahl der relevanten Zeichen. Für Arbeiten auf Strings bietet die Standardbibliothek außerdem viele Funktionen in der Headerdatei `<string.h>` an.

9.3.1 Strings initialisieren

Zur Initialisierung von `char`-Arrays können Sie String-Literale in Anführungszeichen verwenden, anstatt ein Array Zeichen für Zeichen zu initialisieren. Somit wären folgende zwei `char`-Array-Definitionen gleichwertig:

```
00 // kap009/listing007.c
01 #include <stdio.h>
02 #include <stdlib.h>

03 int main(void) {
04   char string1[20] = "String";
05   char string2[20] = {'S', 't', 'r', 'i', 'n', 'g', '\0'};
06   printf("%s\n", string1);
07   printf("%s\n", string2);
08   return EXIT_SUCCESS;
09 }
```

Beide Initialisierungen in den Zeilen **(04)** und **(05)** sind äquivalent. Es wird jeweils ein `char`-Array definiert, das darstellbare 19 Zeichen (!) enthalten kann. Die beiden Strings selbst enthalten davon nur sechs Zeichen. Die restlichen Zeichen werden auch hier, wie schon bei den bisher kennengelernten Arrays, mit 0 vorbelegt. Es wäre im Beispiel allerdings falsch, die Strings mit einer Längenangabe von 6 wie `string1[6]` zu definieren, weil dann kein Platz mehr für das Null-Zeichen übrig wäre. Im Beispiel könnten Sie sehen, wie Sie mit `printf` und der Formatangabe `%s` den kompletten String ausgeben können.

Überlebenswichtiges Stringende-Zeichen

Ein `char`-Array, das einen String speichert, muss immer um mindestens ein Element länger sein als die Anzahl der relevanten (lesbaren) Zeichen. Nur dann kann es noch das Stringende-Zeichen (oder auch Null-Zeichen) `'\0'` aufnehmen. Haben Sie also beispielsweise einen Text mit exakt 10 Zeichen, müssen Sie dafür ein `char`-Array mit 11 Zeichen verwenden. Dieses Stringende-Zeichen ist von enormer Bedeutung bei Stringverarbeitungsfunktionen.

Etwas muss hier jedoch richtiggestellt werden: Es ist **nicht** falsch, wenn Sie bei einem `char`-Array kein abschließendes `'\0'` verwenden. Das gilt allerdings nur dann, wenn Sie die einzelnen Elemente im `char`-Array verwenden wollen. Sobald Sie das `char`-Array als String – also als Ganzes – verwenden wollen, und sei es nur zur Ausgabe auf dem Bildschirm mit `printf`, müssen Sie das Array mit `'\0'` abschließen.

Sie müssen immer zwischen einer Zeichenkonstante und einer Stringkonstante unterscheiden. Folgende Definitionen sind nicht äquivalent:

```
// Zeichenkonstante mit einem Zeichen
char ch = 'X';
// Stringkonstante mit zwei Zeichen: 'X' und '\0'
char ch[] = "X";
```

Natürlich können Sie auch bei den Strings bzw. `char`-Arrays bei der Definition mit der Initialisierungsliste auf die Längenangabe verzichten. Folgende äquivalente Möglichkeiten stehen Ihnen dabei zur Verfügung:

```
char str[] = { 'S', 'T', 'R', 'I', 'N', 'G', '\n', '\0' };
char str[] = "STRING\n";
```

9.3.2 Einlesen von Strings

Zwar wird das Thema Ein-/Ausgabe noch gesondert in Kapitel 14, »Eingabe- und Ausgabe-Funktionen«, behandelt, aber trotzdem soll hier kurz auf die Eingabe von Strings eingegangen werden. Natürlich ist es möglich, `char`-Arrays formatiert mit `scanf` einzulesen. Die `scanf`-Funktion liest allerdings nur bis zum ersten Whitespace-Zeichen ein. Alle restlichen Zeichen dahinter werden somit ignoriert. Außerdem ist `scanf` nicht unbedingt die sicherste Alternative und anfällig für einen Pufferüberlauf (*buffer overflow*), wenn keine oder eine falsche Längenbegrenzung verwendet wird. Eine solche Längenbegrenzung für `scanf` können Sie wie folgt einsetzen:

```
01  char name[20];
02  printf("Bitte Ihren Namen: ");
03  if( scanf("%19s", name) != 1 ) {
04    printf("Fehler bei der Eingabe\n");
05    return EXIT_FAILURE;
06  }
07  printf("Ihr Name ist %s\n", name);
```

In der Zeile **(03)** legen Sie die Längenbegrenzung für die einzulesenden Zeichen auf 19 Zeichen (`%`**`19`**`s`) fest, damit es nicht zu einem Pufferüberlauf kommen kann. Allerdings muss hierbei noch angemerkt werden: Wenn mehr als 19 Zeichen eingegeben wurden, liegen die darüber hinausgehenden Zeichen im Eingabepuffer des Programms. Dies sollten Sie wissen, sofern Sie vorhaben, gleich einen weiteres `scanf` aufzurufen. Der Codeausschnitt bei der Ausführung:

```
Bitte Ihren Namen: Jürgen Wolf
Ihr Name ist Jürgen
```

Wie bereits eingangs erwähnt, liest die Funktion `scanf` nur bis zum ersten Whitespace-Zeichen ein, weshalb in diesem Beispiel bei der Ausführung der Nachname nicht mehr mit eingelesen wird. Führende Whitespace-Zeichen hingegen haben keine Bedeutung.

In den meisten Fällen ist die Standardfunktion `fgets()`die bessere Alternative zum Einlesen von Strings. Die Syntax von `fgets()`:

```
#include <stdio.h>  // Benötigter Header
char *fgets(char *str, int n_chars, FILE *stream);
```

Den String geben Sie mit dem ersten Parameter `str` an. Im zweiten Parameter geben Sie an, wie viele Zeichen eingelesen werden. Das Lesen wird abgebrochen, wenn das Zeilenende `'\n'` oder `nchar_n-1` Zeichen eingelesen wurden. Von wo Sie etwas einlesen wollen, geben Sie mit dem dritten Parameter `stream` an. In unserem Fall sollte es die Standardeingabe sein, die Sie mit dem Stream `stdin` angeben können. Die Funktion `fgets()` kann neben Strings auch zum zeilenweisen Lesen aus Dateien verwendet werden. Die Funktion gibt bei Erfolg die Anfangsadresse auf den eingelesenen String `str`, im Fehlerfall `NULL` zurück. Mehr dazu erfahren Sie in Kapitel 14, »Eingabe- und Ausgabe-Funktionen«.

Hier sehen Sie an einem einfachen Anwendungsbeispiel, wie Sie mit der Funktion `fgets()` Strings einlesen können:

```
00 // kap009/listing008.c
01 #include <stdio.h>
02 #include <stdlib.h>
03 #define MAX 20

04 int main(void) {
05   char string1[MAX];
06   printf("Eingabe machen: ");
07   if (fgets(string1, sizeof(string1), stdin) == NULL ) {
08     printf("Fehler beim Einlesen\n");
09     return EXIT_FAILURE;
10   }
11   printf("Ihre Eingabe: %s", string1);
12   return EXIT_SUCCESS;
13 }
```

In Zeile **(07)** werden mit `fgets()` von der Standardeingabe (`stdin`) maximal `sizeof(string1)` Zeichen in das `char`-Array `string1` eingelesen. Die Funk-

tion `fgets()` garantiert außerdem, dass immer das Stringende-Zeichen an das Ende angefügt wird. Maximal werden immer `sizeof(string1)`-Zeichen an `string1` übergeben. Wenn noch Platz vorhanden ist, wird außerdem das Newline-Zeichen `'\n'` angehängt. Geben Sie im obigen Beispiel einen String mit 20 Zeichen ein, wird kein Newline-Zeichen mehr hinzugefügt, weil das letzte Zeichen dem Null-Zeichen vorbehalten ist. Anstelle der Ausgabe in Zeile **(11)** mit `printf` könnten Sie auch `fputs()`, das Gegenstück von `fgets()`, verwenden.

Das Programm bei der Ausführung:

```
Eingabe machen: Hallo Welt
Ihre Eingabe: Hallo Welt
```

9.3.3 Unicode-Unterstützung

Mit C11 wurde die Unterstützung für Unicode-Zeichen und String-Literale hinzugefügt. Mithilfe verschiedener Präfixe können UTF-8, UTF-16 und UTF-32 verwendet werden. Mit dem Präfix `u8` erstellen Sie einen UTF-8-encodierten String. Ähnliches gilt für die Präfixe `u` und `U`, mit denen Sie UTF-16- bzw. UTF-32- Strings verwenden können.

Ein Beispiel:

```
#include <uchar.h>

/* UTF-8 */
char u8str[] = u8"ΩΩ UTF-8-String ΩΩ";
char u8chr = u8'Ω';


/* UTF-16 */
#ifdef __STD_UTF_16__
    char16_t u16str[] = u"ΩΩ UTF-16-String ΩΩ.";
    char16_t u16char = u'Ω';
#endif

/* UTF-32 */
#ifdef __STD_UTF_32__
```

```
    char32_t u32str[] = U"ΩΩ UTF-32-String ΩΩ";
    char32_t u32char = U'Ω';
#endif
```

`char16_t` und `char32_t` sind ebenfalls seit C11 dabei und eignen sich, um die UTF-16- und UTF-32-kodierten Zeichensequenzen zu speichern. Wie schon bei der Einführung zu `char16_t` und `char32_t` in Abschnitt 3.5.3, »Unicode-Unterstützung«, erwähnt, ist die Verwendung von Unicode-Zeichen keineswegs ein triviales Thema, und C liefert Ihnen hier im Grunde nur ein Fundament. Sie müssen sich als Programmierer in der Regel selbst darum kümmern, dass die richtige Codierung verfügbar ist. Zum Wechseln zwischen den Codierungen bietet Ihnen die Standardbibliothek wiederum Funktionen an. Die Typen und weitere Umwandlungsfunktionen sind in der Headerdatei `<uchar.h>` definiert.

9.3.4 Stringfunktionen der Standardbibliothek – <string.h>

Funktionen, mit denen Sie Strings kopieren, zusammenfügen oder vergleichen können, sind in der Standard-Headerdatei `<string.h>` definiert. Das folgende Beispiel soll die drei häufig verwendeten Funktionen `strncat()` zum Aneinanderhängen, `strncpy()` zum Kopieren und `strncmp()` zum Vergleichen von `char`-Arrays bzw. Strings demonstrieren. Hierbei wird allerdings nicht sehr detailliert auf die einzelnen Funktionen eingegangen. Ich empfehle Ihnen, eine der Online-Referenzen, die Manpages oder das Dokument zum C11-Standard zu Rate zu ziehen.

```
00  // kap009/listing009.c
01  #include <stdio.h>
02  #include <stdlib.h>
03  #include <string.h>
04  #define MAX 50

05  void nl2space( char str[] ) {
06    int n = strlen(str);
07    for(int i = 0; i < n; i++) {
08      if( str[i] == '\n' ) {
09        str[i] = ' ';
```

```
10      }
11    }
12  }

13  void nl2null( char str[] ) {
14    int n = strlen(str)-1;
15    if(str[n] == '\n') {
16      str[n] = '\0';
17    }
18  }

19  int main(void) {
20    char name[MAX*2];
21    char vname[MAX], nname[MAX];
22    printf("Vorname: ");
23    if( fgets(vname, MAX, stdin) == NULL ) {
24      printf("Fehler bei der Eingabe\n");
25      return EXIT_FAILURE;
26    }
27    nl2space( vname );
28    printf("Nachname: ");
29    if( fgets(nname, MAX, stdin) == NULL ) {
30      printf("Fehler bei der Eingabe\n");
31      return EXIT_FAILURE;
32    }
33    nl2null( nname );

34    // Strings vergleichen
35    if( strncmp( vname, nname, MAX ) == 0) {
36      printf("Vorname und Nachname sind identisch\n");
37      return EXIT_FAILURE;
38    }
39    // vname nach name kopieren
40    if( strncpy(name, vname, MAX) == NULL ) {
41      printf("Fehler bei strncpy\n");
42      return EXIT_FAILURE;
43    }
```

```
44    // noch vorhandenen Platz in name ermitteln
45    size_t len = MAX*2 - strlen(name)+1;
46    // nname an name anhängen
47    if( strncat(name, nname, len) == NULL ) {
48      printf("Fehler bei strncat\n");
49      return EXIT_FAILURE;
50    }
51    // gesamten String ausgaben
52    printf("Ihr Name: %s\n", name);
53    return EXIT_SUCCESS;
54  }
```

Hier wurde ein etwas umfangreicheres Beispiel erstellt. Zunächst werden Sie nach dem Vor- und Nachnamen gefragt. Beide werden jeweils mit `fgets()` (in den Zeilen **(23)** und **(29)**) in ein `char`-Array eingelesen. Von beiden Strings wird in den Zeilen **(27)** und **(33)** die Anfangsadresse an die Funktion `nl2space()` bzw. `nl2null()` übergeben, wo ein eventuell vorhandenes Newline-Zeichen von `fgets()` durch ein Leerzeichen (bei `nl2space()`) bzw. Stringende-Zeichen (`nl2null()`) ersetzt werden soll. In den Zeilen **(06)** und **(14)** wird die Funktion `strlen()` verwendet, die ebenfalls in der Headerdatei `<string.h>` definiert ist. Sie gibt die Anzahl der Zeichen eines Strings ohne das Stringende-Zeichen zurück.

In Zeile **(35)** wird überprüft, ob die beiden eingegebenen Strings identisch sind. Ist dies der Fall, gibt die Funktion `strncmp` (ebenfalls als Teil der Standardbibliothek in `<string.h>` enthalten) den Wert 0 zurück, und Sie beenden das Programm mit `EXIT_FAILURE`. In Zeile **(40)** wird der String `vname` in den String `name` mit der Funktion `strncpy()` kopiert. Mit dem dritten Parameter geben Sie an, wie viele Zeichen Sie maximal in `name` kopieren können.

In Zeile **(45)** wird mit der Funktion `strlen` nachgezählt, wie viele Zeichen sich bereits im String `name` befinden, um dann in Zeile **(47)** mittels der Funktion `strncat` maximal `len` Zeichen vom String `nname` an `name` anzuhängen.

Das Programm bei der Ausführung:

```
Vorname: Juergen
Nachname: Wolf
Ihr Name: Juergen Wolf
```

Buffer-Overflow (Pufferüberlauf)

Die beiden Stringfunktionen `strncpy()` und `strncat()` haben jeweils eine Schwesterfunktion mit `strcpy()` und `strcat()` ohne das `n` (das für **n**umerable steht) im Namen. Zwar haben diese Versionen nur zwei Parameter und lassen sich einfacher verwenden, aber sie verursachen auch das Problem, dass nicht auf die Größe des Zielstrings geachtet wird. So kann bei falscher Verwendung ein Pufferüberlauf (Buffer-Overflow) ausgelöst und von Hackern allerlei Unfug auf dem System angerichtet werden.

9.3.5 Sicherere Funktionen zum Schutz vor Speicherüberschreitungen

Gerade bezüglich der Stringfunktionen (und auch anderen Bibliotheksfunktionen) gibt es seit dem C11-Standard sicherere im Annex K des C11-Standard beschriebene optionale Erweiterungen, um Speicherüberschreitungen (engl. *bounds checking*) zu reduzieren. Dabei handelt es sich um bekannte Funktionen der C-Standardbibliothek, denen die Endung `_s` hinzugefügt wurde. Die *bounds-checking*-Versionen von `strcat()` und `strncpy()` lauten dann beispielsweise `strcat_s()` und `strncpy_s()`. Ob Annex K überhaupt unterstützt wird, können Sie über das Makro `__STDC_LIB_EXT1__` testen. Ist `__STDC_LIB_EXT1__` definiert, dann wurde die Bibliothek standardkonform nach Annex K implementiert.

An dieser Stelle sollte auch noch hinzugefügt werden, dass viele Funktionen wie `strncpy()` oder `strncat()` nicht direkt als »unsicher« gelten, wenn man ihre Funktion und Besonderheiten kennt und beachtet.

9.3.6 Umwandlungsfunktionen zwischen Zahlen und Strings

Sollten Sie auf der Suche nach Funktionen sein, mit denen Sie einen String in einen numerischen Wert konvertieren können, werden Sie in der Headerdatei `<stdlib.h>` fündig. Hier finden Sie z. B. die Möglichkeit, einen String mit der Funktion `strtod()` in einen `double`-Wert oder mit `strtol()`in einen `long`-Wert zu konvertieren.

9.4 Kontrollfragen und Aufgaben

1. Was sind Arrays?
2. Wo liegt der Unterschied zwischen Strings und Arrays?
3. Was ist die größte Gefahr bei der Verwendung von Arrays bzw. Strings?
4. Welche Indexnummer hat das erste Element eines Arrays oder Strings?
5. Im folgenden Listing wurden gleich zwei Fehler gemacht. Welche?

```
00 // kap009/aufgabe001.c
01 #include <stdio.h>
02 #include <stdlib.h>
03 #define MAX 10

04 int main(void) {
05   int ival[MAX];
06   for(int i = MAX; i > 0; i--) {
07     ival[i] = i;
08   }
09   for(int i = 0; i < MAX; i++) {
10     printf("%d\n", ival[i]);
11   }
12   return EXIT_SUCCESS;
13 }
```

6. Auch wenn das folgende Programm korrekt ausgeführt wird, ist ein Fehler enthalten. Welcher?

```
00 // kap009/aufgabe002.c
01 #include <stdio.h>
02 #include <stdlib.h>
03 #define MAX 10

04 int main(void) {
05   char v[5] = { 'A', 'E', 'I', 'O', 'U' };
06   printf("Die einzelnen Vokale: ");
07   for(int i=0; i < 5; i++ ) {
08     printf("%c (Dezimal: %d)\n", v[i], v[i]);
09   }
```

```
10    printf("Alle zusammen: %s\n", v);
11    return EXIT_SUCCESS;
12  }
```

7. Schreiben Sie ein Programm, das die Größe in Bytes und die Anzahl der Elemente eines Arrays bzw. Strings ermittelt und ausgibt. **Tipp:** Verwenden Sie den `sizeof`-Operator. Bei den Strings können Sie auch die Funktion `strlen()` aus der Headerdatei `<string.h>` nutzen. Verwenden Sie folgende Arrays bzw. Strings:

   ```
   int iarr[]    = { 2, 4, 6, 4, 2, 4, 5, 6, 7 };
   double darr[] = { 3.3, 4.4, 2.3, 5.8, 7.7 };
   char str[]    = { "Hallo Welt"};
   ```

8. Schreiben Sie eine Funktion, die zwei `int`-Arrays auf Gleichheit überprüft. Die Funktion soll –1 zurückgeben, wenn beide Arrays gleich sind, oder die Position, an der ein Unterschied gefunden wurde. –2 soll zurückgegeben werden, wenn beide Arrays unterschiedlich lang sind. **Hinweis:** Verwenden Sie hierfür **nicht** die Funktion `memcmp()` aus der Headerdatei `<string.h>`, mit der Sie ebenfalls zwei Arrays auf Gleichheit überprüfen könnten.
9. Schreiben Sie eine Funktion, die in einem String ein bestimmtes Zeichen durch ein anderes Zeichen ersetzt.

Kapitel 10

Zeiger (Pointer)

In diesem Kapitel soll eines der wichtigsten Themen in C behandelt werden: die Zeiger. Sie werden auch Pointer genannt. Haben Sie die Zeigerarithmetik erst einmal verstanden, sind auch fortgeschrittene Themen keine so große Hürde mehr. Es folgt ein kleiner Überblick, was mit den Zeigern u. a. alles realisiert wird:

- Speicherbereiche können dynamisch zur Laufzeit reserviert, verwaltet und wieder gelöscht werden.
- Mit Zeigern können Sie Datenobjekte per Referenz an Funktionen übergeben.
- Mit Zeigern lassen sich Funktionen als Argumente an andere Funktionen übergeben.
- Komplexe Datenstrukturen wie Listen und Bäume lassen sich ohne Zeiger gar nicht realisieren.
- Es lässt sich ein typenloser Zeiger (`void *`) definieren, mit dem Datenobjekte beliebigen Typs verarbeitet werden können.

10.1 Zeiger vereinbaren

Wenn Sie verstehen, dass Zeiger lediglich die Adresse und den Typ eines Speicherobjekts darstellen, haben Sie das Wichtigste verstanden. Die Definition eines solchen Zeigers sieht wie folgt aus:

```
Datentyp *name;
```

Am Stern `*` zwischen `Datentyp` und `name` können Sie den Zeiger erkennen. Der `name` ist der Bezeichner und wird als Zeiger auf einem Typ `Datentyp` deklariert. Für `name` gelten alle üblichen Regeln, die schon in Abschnitt 2.4.1, »Bezeichner«, beschrieben wurden. Zusätzlich können bei den Zeigern

auch noch die Typ-Qualifizierer `const`, `volatile` oder `restrict` verwendet werden.

Zeiger zeigen wohin?

Bei der Verwendung von Zeigern ist häufig die Rede von »zeigen auf«. Dies hilft ungemein, das Thema besser zu verstehen. Ein Rechner kann natürlich nicht im bildlichen Sinne »zeigen«. Wenn Sie also »auf etwas zeigen« lesen, ist damit gemeint, dass auf einen bestimmten Speicherbereich, also eine Adresse im Arbeitsspeicher referenziert wird.

Ein einfaches Beispiel:

```
int *iptr;
```

Hier wurde ein Zeiger mit dem Namen `iptr` erstellt, der auf ein Speicherobjekt vom Typ `int` verweisen kann. Noch genauer gesagt, kann dieser die Adresse eines `int`-Objekts speichern.

10.2 Zeiger verwenden

Wird im Programm ein Zeiger mit automatischer Speicherdauer (innerhalb eines Blocks ohne das Schlüsselwort `static`) verwendet, der zuvor nicht initialisiert wurde, kann dies zu schwerwiegenden Fehlern führen. Ein Zeiger, der nicht mit einer gültigen Adresse initialisiert wurde und auf den jetzt zurückgegriffen werden soll, greift stattdessen nämlich einfach auf irgendeine Adresse im Arbeitsspeicher zurück. Wenn sich in diesem Speicherbereich wichtige Daten oder Programme in der Ausführung befinden, kommt es logischerweise zu Problemen.

Automatische Zeiger haben innerhalb eines Blocks ohne eine Initialisierung einen undefinierten Anfangswert. Globale bzw. `static`-Zeiger werden ohne einen Initialisierer mit einem `NULL`-Zeiger implizit initialisiert.

Um beispielsweise einen Zeiger vom Typ `int` auf die Adresse einer `int`-Variablen verweisen zu lassen, müssen Sie folgende Zuweisung darauf erstellen:

```
int *iptr;                  // Ein int-Zeiger
int ival = 255;             // Eine int-Variable
iptr = &ival;               // iptr enhält die Addresse von ival
```

Mithilfe des Adressoperators `&` wurde dem Zeiger `iptr` die Adresse der Variablen `ival` zugewiesen. Abbildung 10.1 stellt den Vorgang grafisch dar.

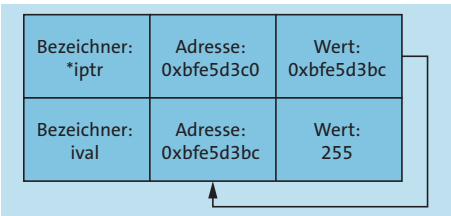

Abbildung 10.1 Zeiger verweisen auf die Adressen von anderen Speicherobjekten.

Das Formatzeichen `%p` im folgenden Beispiel trägt zum besseren Verständnis bei. Hierzu ein kleines Beispiel:

```
00  // kap010/listing001.c
01  #include <stdio.h>
02  #include <stdlib.h>

03  int main(void) {
04    int *iptr;
05    int ival = 255;
06    iptr = &ival;
07    printf("Adresse iptr: %p\n", &iptr);
08    printf("zeigt auf   : %p\n", iptr);
09    printf("Adresse ival: %p\n", &ival);
10    return EXIT_SUCCESS;
11  }
```

Vor der Beschreibung das Programm bei der Ausführung:

```
Adresse iptr: 000000000013ff00
zeigt auf   : 000000000013ff0c
Adresse ival: 000000000013ff0c
```

In Zeile **(06)** des Listings wurde dem Zeiger `iptr` die Adresse der Variablen `ival` zugewiesen. In Zeile **(07)** wird mithilfe des Formatzeichens `%p` die Adresse des Zeigers `iptr` im Arbeitsspeicher ausgegeben. Auch hierfür müssen Sie den Adressoperator verwenden. In Zeile **(08)** wird hingegen die Adresse ausgegeben, auf die der Zeiger `iptr` verweist. Dass dies in dieser Zeile ohne den Adressoperator `&` funktioniert, liegt natürlich daran, dass ein Zeiger selbst auch nur Adressen speichert, auf die er referenziert. In Zeile **(09)** wird dieselbe Adresse wie in Zeile **(08)** ausgegeben, weil Sie in Zeile **(06)** den Zeiger `iptr` auf die Adresse der Variablen `ival` verwiesen haben. Natürlich müssen Sie bei einer Variablen wieder den Adressoperator verwenden, wenn Sie an der Adresse und nicht dem Wert der Variablen interessiert sind.

Explizite Typenumwandlung für byteweisen Zugriff

In speziellen Fällen ist es nötig, den Wert eines Zeigers explizit in einen anderen Typ umzuwandeln. Wollen Sie beispielsweise ein Speicherobjekt Byte für Byte auslesen, wird dafür gewöhnlich ein `char`-Zeiger verwendet. Hier ein Beispiel:

```
char *bytePtr;
float fval = 255.255;
bytePtr = (char *)&fval;
```

Mit der expliziten Umwandlung zeigt jetzt ein `char`-Zeiger auf das erste Byte des Speicherobjekts `fval` und könnte somit Byte für Byte bearbeitet werden.

10.3 Zugriff auf den Inhalt von Zeigern

Sie verwenden den Indirektionsoperator *, um direkt auf die Werte eines Speicherobjekts mit einem Zeiger zuzugreifen, auf das Sie zuvor mit dem Adressoperator referenziert haben. Der direkte Zugriff auf den Wert eines Speicherobjekts mit dem Indirektionsoperator (oder auch Verweisoperator) wird häufig auch als *Dereferenzierung* bezeichnet. Sie können damit über einen Zeiger die Werte, auf die er verweist, auslesen oder ändern. Hier ein einfaches Beispiel:

```
00 // kap010/listing002.c
01 #include <stdio.h>
02 #include <stdlib.h>

03 int main(void) {
04   int *iptr;
05   int ival = 255;
06   iptr = &ival;
07   // Wert ausgeben
08   printf("*iptr  :  %d\n", *iptr);
09   printf(" ival  :  %d\n", ival);
10   // ival neuen Wert zuweisen
11   *iptr = 128;
12   // Wert ausgeben
13   printf("*iptr  :  %d\n", *iptr);
14   printf(" ival  :  %d\n", ival);
15   return EXIT_SUCCESS;
16 }
```

Das Programm bei der Ausführung:

```
*iptr  :  255
 ival  :  255
*iptr  :  128
 ival  :  128
```

In Zeile **(08)** wurde der Indirektionsoperator `*` mit dem Zeiger `iptr` verwendet. Im Gegensatz zu `iptr` ohne Indirektionsoperator, der die Adresse des Speicherobjekts `ival` enthält, können Sie mit dem Indirektionsoperator direkt auf den Wert von `ival` zugreifen. In Zeile **(11)** wurde daher zum Beweis ein neuer Wert an `ival` über den Zeiger `iptr` zugewiesen, wie die Ausgabe der Zeilen **(13)** und **(14)** zeigt. Das bedeutet also, dass alle gültigen Integerwerte, die Sie über den Indirektionsoperator `*` und den Zeiger `iptr` zuweisen, eigentlich die Variable `ival` betreffen. Abbildung 10.2 zeigt den Vorgang nochmals bildlich.

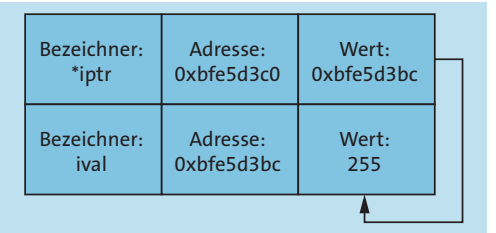

Abbildung 10.2 Mit dem Indirektionsoperator * greifen Sie direkt auf den Wert eines Speicherobjekts zu, auf dessen Adresse der Zeiger verweist.

Stern bei der Zeigerdeklaration und dem Indirektionsoperator

An dieser Stelle soll ein häufiger Irrtum beseitigt werden. Der Stern bei der Deklaration eines Zeigers ist nicht mit dem Indirektionsoperator für die Dereferenzierung zu verwechseln, der auch die Form eines Sterns hat.

Dasselbe funktioniert natürlich auch anders herum. Sie können einer normalen Variablen über den Indirektionsoperator den Wert einer Variablen zuweisen, deren Adresse der Zeiger enthält. Ein einfaches Listing dazu:

```
00  // kap010/listing003.c
01  #include <stdio.h>
02  #include <stdlib.h>

03  int main(void) {
04    float *fptr;
05    float fval1 = 123.123f, fval2;
06    // fptr die Adresse von fval1 übergeben
07    fptr =& fval1;
08    // fval2 erhält den Wert von fval1
09    fval2 = *fptr;
10    printf("fval2: %.3f\n", fval2);
11    return EXIT_SUCCESS;
12  }
```

In Zeile **(07)** erhält der Zeiger `fptr` die Adresse von `fval1`. In Zeile **(09)** wird der Wert dieser Variablen über den Zeiger `fptr` mithilfe des Indirektionsoperators an die Variable `fval2` verwiesen.

NULL-Zeiger

Den Indirektionsoperator dürfen Sie natürlich nur verwenden, wenn der Zeiger eine gültige Adresse enthält. Wurde dem Zeiger keine gültige Adresse zugewiesen und wird der Indirektionsoperator trotzdem verwendet, wird das Programm vermutlich aufgrund einer Speicherzugriffsverletzung (*segmentation fault*) abstürzen oder schlimmsten Fall sogar weiterlaufen.

In der Praxis können Sie viele Fehler vermeiden, wenn Sie einen nicht verwendeten Zeiger zunächst immer mit `NULL` gleich bei der Definition initialisieren und einen Zeiger vor jeder Verwendung überprüfen. `NULL` ist ein Zeiger, wenn keine gültige Adresse verwendet wird. Globale Zeiger oder Zeiger, die mit `static` gekennzeichnet wurden, werden automatisch mit dem `NULL`-Zeiger initialisiert. Lokale Zeiger in einem Anweisungsblock enthalten dagegen ein beliebiges und somit undefiniertes Bitmuster.

Es folgt ein Beispiel mit dem `NULL`-Zeiger. Mit diesem können Sie eine Speicherzugriffsverletzung vermeiden:

```
00  // kap010/listing003-null.c
01  #include <stdio.h>
02  #include <stdlib.h>

03  int main(void) {
04    int *iptr = NULL;  // Zeiger mit NULL initialisiert
05    // ... mehr Code
06    if( iptr == NULL ) {
07      printf("Zeiger hat keine gueltige Adresse\n");
08      return EXIT_FAILURE;
09    }
10    // iptr hat eine gültige Adresse ...
11    return EXIT_SUCCESS;
12  }
```

In Zeile **(04)** wurde dem Zeiger `iptr` bei der Initialisierung gleich der `NULL`-Zeiger zugewiesen. Einen Zeiger sollten Sie immer auf eine gültige Adresse hin prüfen. Dies geschieht in Zeile **(06)**. Bei einem Fehler wird das Programm beendet, oder es werden andere Vorkehrungen getroffen.

»NULL«-Zeiger

Der `NULL`-Zeiger ist ein vordefinierter Zeiger, gewöhnlich mit einem Wert auf die Adresse 0. In C werden Variablen und Funktionen immer an bestimmten Speicherplätzen abgelegt, deren Adresse unterschiedlich von 0 ist. Daher ist ein Zeiger auf die Adresse 0 ein Zeiger auf ein ungültiges Speicherobjekt. Die Konstante `NULL` ist in der Headerdatei `<stddef.h>` definiert.

Deklaration, Adressierung und Dereferenzierung von Zeigern

Sie dürften nun schon gemerkt haben, warum das Thema Zeiger etwas komplexer ist. Schwierig ist allerdings eher selten das Verständnis der Zeiger selbst, welche ja letztendlich nur mit Adressen operieren, sondern die richtige Verwendung des Adressoperators und des Indirektionsoperators. Daher hier eine Übersicht über den Zugriff und die Dereferenzierung von Zeigern:

```
int *iptr = NULL; // Zeiger mit NULL initialisiert
int ival1=0, ival2=0;

// Initialisierung: Zeiger erhält Adresse von ival1
iptr = &ival1;

// Dereferenzierung mit dem Indirektionsoperator
// ival erhält den Wert 123
*iptr = 123;

// ival2 erhält denselben Wert wie ival1
ival2 = *iptr;
```

```
// ival erhält die Summe aus ival2 + 100;
*iptr = ival2+100;

// Zeiger erhält die Adresse von ival2
iptr = &ival2;

printf("%d\n" , *iptr);  // gibt Wert von ival2 aus
printf("%p\n" , iptr);   // gibt die Adresse von ival2 aus
printf("%p\n" , &ival2); // gibt die Adresse von ival2 aus
printf("%p\n" , &iptr);  // gibt die Adresse von iptr aus
```

10.4 Zeiger als Funktionsparameter

Funktionen, die mit einem oder mehreren Parametern definiert werden und mit `return` einen Rückgabewert zurückliefern, haben wir bereits verwendet. Hierbei wurden bei jedem Aufruf alle Parameter kopiert, sodass diese Variablen der Funktion anschließend als lokale Variablen zur Verfügung stehen.

Als Alternative bietet es sich an, die Adressen der entsprechenden Variablen – statt einer Kopie – an die Funktion zu übergeben. Und wenn von Adressen die Rede ist, sind die Zeiger nicht weit entfernt.

```
00  // kap010/listing004.c
01  #include <stdio.h>
02  #include <stdlib.h>

03  void reset( int *val ) {
04    *val = 0;
05  }

06  int main(void) {
07    int ival = 1234567;
08    printf("ival: %d\n", ival);   // = 1234567
09    reset( &ival );
```

```
10    printf("ival: %d\n", ival);   // = 0
11    return EXIT_SUCCESS;
12  }
```

In Zeile **(09)** übergeben Sie mit dem Adressoperator die Adresse der Variablen `ival` an die Funktion `reset()`. Im Funktionskopf `reset()` der Zeile **(03)** muss natürlich als formaler Parameter ein Zeiger mit dem entsprechenden Typ definiert sein. Durch den Aufruf der Zeile **(09)** wird also dem Zeiger `val` in Zeile **(03)** die Adresse der Variablen `ival` zugewiesen. Mithilfe des Indirektionsoperators und `val` können Sie jetzt auf den Wert der Variablen `ival` zugreifen. In Zeile **(04)** wurde der Variablen `ival` der Einfachheit halber indirekt der Wert `0` zugewiesen. Dies erfolgte über den Zeiger `val` mit dem Indirektionsoperator.

Es ist Ihnen sicherlich aufgefallen, dass bei der Funktion keine Rückgabe mit `return` erfolgt und der Rückgabetyp `void` ist. Eine Rückgabe wird hier nicht mehr benötigt, da wir den Wert direkt in die Variable des Aufrufers schreiben.

10.5 Zeiger als Rückgabewert

Ein Zeiger kann auch als Rückgabetyp einer Funktion deklariert werden. Dann gibt er logischerweise auch nur die Anfangsadresse des Rückgabewerts zurück. Die Syntax dazu sieht folgendermaßen aus:

```
Typ* Funktionsname( formale Parameter )
```

Das Verfahren mit Zeigern als Rückgabewert von Funktionen wird bei Arrays, Strings oder Strukturen verwendet und ist eine effiziente Methode, umfangreiche Datenobjekte aus einer Funktion zurückzugeben. Natürlich werden hier nicht ganze Datenobjekte, sondern nur die Anfangsadresse darauf an den Aufrufer zurückgegeben.

Wenn Sie sich noch an Abschnitt 7.6, »Exkurs: Funktion bei der Ausführung«, erinnern, wissen Sie, dass beim Aufruf einer Funktion ein Stack verwendet wird. Auf diesem werden alle benötigten Daten einer Funktion (die Parameter, die lokalen Variablen und die Rücksprungadresse) ange-

legt. Die Rede ist vom Stack-Frame. Dieser bleibt so lange bestehen, bis sich die Funktion wieder beendet. Geben Sie eine Adresse auf einen solchen Speicherbereich (lokalen Speicher) zurück, der sich ebenfalls auf dem Stack-Frame befand, und somit bei Beendigung der Funktion nicht mehr vorhanden ist, wird ein undefinierter Speicherbereich zurückgegeben.

Wollen Sie also einen Zeiger auf einen gültigen Speicherbereich zurückgeben, haben Sie folgende Möglichkeiten. Sie verwenden

- einen statischen (`static`) oder einen globalen Speicher,
- einen beim Aufruf der Funktion als Argument übergebenen Speicher (bzw. eine Adresse) oder
- einen mittels `malloc()` zur Laufzeit reservierten Speicher (siehe Kapitel 11, »Dynamische Speicherverwaltung«).

In der Praxis würde ich zwar die dynamische Reservierung von Speicher zur Laufzeit mit `malloc()` empfehlen, aber um nicht auf dieses Thema vorzugreifen, soll hier ein Beispiel mit einem statischen Speicher demonstriert werden:

```
00 // kap010/listing005.c
01 #include <stdio.h>
02 #include <stdlib.h>
03 #define MAX 5

04 int* ifget( void ) {
05   static int puffer[MAX];
06   for(int i=0; i<MAX; i++) {
07     printf("Wert %d : ", i+1);
08     if( scanf("%d", &puffer[i] ) != 1 ) {
09       printf("Fehler bei der Eingabe\n");
10       return EXIT_SUCCESS;
11     }
12   }
13   return puffer;
14 }

15 int main(void) {
```

```
16    int* iptr = ifget( );
17    printf("Folgende Werte wurden eingelesen\n");
18    for(int i=0; i < MAX; i++ ) {
19      printf("%d : %d\n", i+1, *(iptr+i));
20    }
21    return EXIT_SUCCESS;
22  }
```

In Zeile **(16)** wird die Funktion `ifget()` aufgerufen. Der Rückgabewert wird an den `int`-Zeiger `iptr` zugewiesen. Entsprechend muss natürlich auch der Funktionskopf der Zeile **(04)** aufgebaut sein, die einen Zeiger vom Typ `int*` zurückgibt. Damit die Funktion einen gültigen Speicherbereich zurückgibt, wurde in Zeile **(05)** mit dem Schlüsselwort `static` ein statischer Speicher, ein Array, mit `MAX` `int`-Werten definiert. Diesem Array übergeben Sie in der `for`-Schleife (Zeile **(06)** bis **(12)**) insgesamt `MAX` `int`-Werte. In Zeile **(13)** geben Sie die Anfangsadresse des statischen Speicherbereichs an den Aufrufer der Zeile **(16)** zurück. In der `main()`-Funktion werden die in der Funktion `ifget()` eingegebenen `int`-Werte nochmals in einer `for`-Schleife in den Zeilen **(18)** bis **(20)** ausgegeben.

Würden Sie das Schlüsselwort `static` in Zeile **(05)** entfernen, würden in der `main()`-Funktion undefinierte Werte ausgegeben, weil ein lokaler Speicher nach dem Ende der Funktion, der nicht mehr vorhanden ist, an den Aufrufer zurückgegeben und verwendet würde. Abhängig vom Compiler und der Warnstufe sollten Sie allerdings auch eine entsprechende Warnung erhalten.

Das Programm bei der Ausführung:

```
Wert 1 : 12
Wert 2 : 34
Wert 3 : 56
Wert 4 : 78
Wert 5 : 90
Folgende Werte wurden eingelesen
1 : 12
2 : 34
```

```
3 : 56
4 : 78
5 : 90
```

10.6 Zeigerarithmetik

Bei der Zeigerarithmetik (bzw. auch Pointer-Arithmetik) ist die Rede vom Zugriff auf die Zeiger ohne den Indirektionsoperator *. Es geht also rein um die Verwendung von Operatoren für Zeiger und nicht um deren Werte bzw. Objekte, auf die sie zeigen. In der Praxis werden Sie die Zeigerarithmetik häufig in Verbindung mit den C-Arrays verwenden. Folgende Operationen sind mit den Zeigern erlaubt:

- **Vergleiche zweier Zeiger** (`zeiger1` *op* `zeiger2`) mit folgenden Operatoren für *op*: `==`, `!=`, `<`, `<=`, `>` und `>=`. Die Verwendung von Vergleichsoperatoren ist allerdings nur dann sinnvoll, wenn die Zeiger auf Arrayelemente zeigen. Beide Zeiger müssen außerdem vom selben Typ bzw. einer der Zeiger kann der `NULL`-Zeiger sein. So liefert beispielsweise ein Vergleich von `zeiger1 > zeiger2` wahr zurück, wenn die Speicheradresse von `zeiger1` höher als die Adresse von `zeiger2` ist. Auch wird ein Vergleich oft benutzt, um zu bestimmen, ob ein Zeiger ein `NULL`-Zeiger ist.
- **Subtraktion zweier Zeiger** (`zeiger2 - zeiger1`) mit dem Operator `-`. Als Ergebnis der Subtraktion von `zeiger2 – zeiger1` wird die Anzahl der Elemente zwischen den Zeigern zurückgegeben. Hierfür finden Sie in der Headerdatei `<stddef.h>` den Typ `ptrdiff_t` definiert. Die Subtraktion von zwei Zeigern, die nicht auf dasselbe Element zeigen, ist ein logischer Fehler.
- **Addition und Subtraktion eines Zeigers mit einer Ganzzahl** (`zeiger` *op* `integer`) mit den Operatoren `+`, `-`, `+=`, `-=` für *op* und den Inkrement- bzw. Dekrement-Operatoren `++` und `--`. Zeigt beispielsweise der Zeiger `zeiger1` auf das Arrayelement `array[i]`, bedeutet eine Zuweisung des Zeigers mit `zeiger2=zeiger1+2`, dass `zeiger2` auf das Arrayelement `array[i+2]` zeigt. Auch das Inkrementieren und Dekrementieren von Zeigern mit `++` und `--` ist möglich.

Zuweisungen mit Zeigern

An dieser Stelle sollten noch ein paar Besonderheiten zu den Zuweisungen mit Zeigern genannt werden. Sie können einen Zeiger vom Typ `void*` jederzeit an den Zeiger eines anderen Datentyps zuweisen. Auch können Sie einem Zeiger eines beliebigen Datentyps den Zeiger `void*` zuweisen. Und ebenso können Sie einen `NULL`-Zeiger (da gewöhnlich als `(void*)0` definiert) jedem anderen Zeiger zuweisen. Zeiger mit unterschiedlichen Datentypen dürfen allerdings nicht einander zugewiesen werden!

10.7 Zugriff auf Arrayelemente über Zeiger

Es gibt zwei Möglichkeiten, wie Sie auf ein Arrayelement zugreifen können. Entweder gehen Sie den bereits bekannten Weg über den Index mit `[]`, oder Sie verwenden hierzu die Zeiger in einer Zeiger-Versatz-Schreibweise.

Sehen Sie sich hierzu folgendes Listing an:

```
00  // kap010/listing006.c
01  #include <stdio.h>
02  #include <stdlib.h>
03  #define MAX 5

03  int main(void) {
04    int iarr[MAX] = { 12, 34, 56, 78, 90 };
05    int *iptr = NULL;
06    // iptr zeigt auf das erste Element von iarr
07    iptr = iarr;

08    printf("iarr[0] = %d\n", *iptr);
09    printf("iarr[2] = %d\n", *(iptr+2));
10    *(iptr+4) = 66;   // iarr[4] = 66
11    *iptr = 99;       // iarr[0] = 99
12    // Alle auf einmal durchlaufen (2 Möglichkeiten)
13    int *iptr2 = iarr;  // iptr2 auf den Anfang von iarr
```

```
14    for(int i=0; i < MAX; i++, iptr2++) {
15      printf("iarr[%d] = %d /", i, *(iptr+i));
16      printf(" %d\n", *iptr2);  // so geht es auch
17    }
18    return EXIT_SUCCESS;
19  }
```

In Zeile **(07)** wird der Zeiger `iptr` mit der Adresse des ersten Elements von `iarr` initialisiert. Sicherlich fragen Sie sich, warum in der Zeile

```
iptr = iarr;
```

anstatt

```
iptr = &iarr[0]
```

verwendet wurde. Hier gilt, dass der Name eines Arrays ohne Index automatisch eine Adresse auf das erste Element ist. Der Beweis, dass `iptr` auf das erste Element von `iarr`, genauer `iarr[0]`, verweist, wird in Zeile **(08)** ausgegeben.

In der Zeile **(09)** kommt eine Zeigerarithmetik mit der Addition einer Ganzzahl zum Einsatz, bei welcher der Zeiger bei einer Erhöhung um die Anzahl der Bytes verschoben wird, die der Größe des durch den Zeiger referenzierten Typs entspricht. Die Zeiger-Versatz-Schreibweise `*(iptr+2)` ist identisch zu `iarr[2]`. Mit beiden Versionen wird auf den Inhalt des dritten Elements im Array `iarr` zugegriffen.

Spätestens jetzt dürften Sie auch erkennen, warum Sie Zeiger richtig typisieren müssen. Denn ohne das Wissen um die Speichergröße des assoziierten Typs könnte das Vorgänger- bzw. Nachfolgerelement im Array beispielsweise über die Anweisung `*(iptr+2)` nicht wie in Zeile **(09)** berechnet werden. Im Beispiel wird der Typ `int` verwendet. Daher erfolgt eine Adressierung mit `iptr+n` um die Größe des Typs, auf die der Zeiger verweist. Wird also ein Zeiger vom Typ `int*` um 1 erhöht, so verweist er auf ein `int`-Objekt weiter. Dasselbe gilt beispielsweise auch, wenn ein Zeiger vom Typ `double*`, der auf `double` zeigt, um 1 erhöht wird, sodass dieser Zeiger um die Anzahl der Bytes (`=sizeof(double)`) verschoben wird, welche der Größe des durch den Zeiger referenzierten Typs entspricht.

Zeiger unterschiedlicher Datentypen nicht vermischen!

Wenn Sie im Beispiel einen `double`-Zeiger auf das `int`-Array verweisen lassen, was ja durchaus mit einem hier nicht ratsamen und gewaltsamen expliziten Cast erzwungen werden könnte, würde die Zeigerarithmetik nicht mehr richtig arbeiten: Es wird dann eine Erhöhung der Speicherzelle, auf die der Zeiger verweist, gemäß dem Zeiger und nicht dem Typ der Variablen durchgeführt. Wenn dann etwa `sizeof(double)` gleich 8 und ein `sizeof(int)` gleich 4 wäre, würde bei einer Erhöhung um 1 der Zeiger um 8 Bytes weiterlaufen. Daher sollte es jetzt einleuchten, wie auch in Abschnitt 10.6, »Zeigerarithmetik«, bereits erwähnt, dass Zeiger mit verschiedenen Datentypen nicht einander zugewiesen werden dürfen.

Damit der Zeiger tatsächlich auf die nächste Adresse zeigt, wurde `ptr+n` zwischen Klammern gestellt, weil Klammern eine höhere Bindungskraft haben und somit zuerst ausgewertet werden.

In der `for`-Schleife der Zeilen **(14)** bis **(17)** finden Sie außerdem neben der Möglichkeit, die Adresse des Zeigers der Speicherzeile mit einem ganzzahligen Wert zu erhöhen, auch die Möglichkeit, die einzelnen Adressen des Arrays mit dem Inkrementoperator (`iptr2++`) zu erhöhen und so durch das Array zu iterieren.

Reduzieren von Zeigern

Neben einer Addition von Zeigern mit `ptr+1` können Sie diese ebenfalls mit `ptr-1` subtrahieren. Gleiches gilt auch für den Dekrementoperator `ptr--`.

Um also auf ein Element eines Arrays über Zeiger zuzugreifen, haben Sie folgende Möglichkeiten:

```
01  int iarr[MAX] = { 11, 22, 33, 44, 55 };
02  int *iptr = iarr;
03  // Zugriff auf eine Element
04  printf("%d : %d\n", iarr[2], *(iptr+2));
05  // Zugriff auf Adresse
```

```
06  printf("%p : %p\n", &iarr[3], iptr+3);
07  // Arrayname als Zeiger verwenden
08  printf("%d : %d\n", iarr[1], *(iarr+1));
09  // Zeiger indizieren
10  printf("%d : %d\n", iarr[4], iptr[4]);
```

Beide in der Zeile **(04)** demonstrierten Möglichkeiten, auf ein Element im Array zuzugreifen, sind gleichwertig. In der Zeile **(06)** sehen Sie hingegen zwei gleichwertige Methoden, auf die Adressen (!) eines Arrays zuzugreifen. In der Zeile **(08)** können Sie sehr schön sehen, dass ein Arrayname auch als Zeiger aufgefasst werden kann, und die Zeile **(10)** zeigt den umgekehrten Fall, in dem ein Zeiger mit der Zeiger-Index-Schreibweise indiziert wurde.

Anhand dieser Beispiele können Sie erkennen, dass Arrays und Zeiger recht eng miteinander zusammenhängen. Ein Arrayname ist im Grunde ein konstanter Zeiger. Die Zeiger hingegen können verwendet werden, um auf die Arrayelemente zuzugreifen.

10.8 Array und Zeiger als Funktionsparameter

Das Thema wurde bereits in Abschnitt 9.1.4, »Arrays an Funktionen übergeben«, behandelt und soll hier daher nur noch einmal kurz aufgegriffen werden. Da Arrays nicht als Ganzes direkt an Funktionen übergeben werden können, wird immer nur die Adresse an ein Array übergeben. Daher sind die folgenden zwei Deklarationen der formalen Parameter einer Funktion völlig gleichbedeutend, weil der Compiler im Grunde eben nur an der Adresse des ersten Arrayelements interessiert ist:

```
void modArr(int arr[], size_t s);
// und
void modArr(int *arr, size_t s);
```

Sie können solche Funktionen auf mehrere Arten aufrufen. Allerdings wird immer die (Anfangs-)Adresse des Arrays an die Funktion übergeben, damit diese weiß, wo sich das Array im Speicher befindet. Auf diese Weise bekommt die aufrufende Funktion auch einen direkten Zugriff auf die Daten und kann diese daher manipulieren.

In der Regel wird außerdem bei der Übergabe eines Arrays an eine Funktion auch die Anzahl der Elemente als weiteres Argument mit übergeben, damit in der Funktion die korrekte Anzahl von Elementen im Array abgearbeitet werden kann.

const-Arrayparameter

In diesem Zusammenhang finden Sie häufig Arrayparameter, die mit dem Qualifizierer `const` ausgezeichnet sind. Mit einem `const`-Arrayparameter verhindern Sie Modifikationen der Arraywerte der aufrufenden Funktion. Solche Arrayelemente sind dann in der Funktion konstant, und eine versehentliche Änderung der Daten wird verhindert. Ein Beispiel mit zwei gleichbedeutenden Funktionsprototypen:

```
void printArr(const int arr[], size_t s);
// gleichwertig:
void printArr(const int *arr, size_t s);
```

Eine Änderung der Elemente im Array `arr` ist in diesem Beispiel nicht mehr möglich, weil es als `const` qualifiziert ist. Eine Zuweisung wie

```
arr[0] = 1234; // Fehler in printArr, da const
```

innerhalb der Funktion ist nicht mehr möglich und wird vom Compiler mit einer Fehlermeldung verweigert. In der Praxis ist es daher guter Stil, ein Array mit `const` zu qualifizieren, wenn die Funktion ohnehin keine Manipulation daran vornimmt.

Hierzu noch ein komplettes Beispiel zur Übergabe von Arrays an Funktionen:

```
00 // kap010/listing007.c
01 #include <stdio.h>
02 #include <stdlib.h>

03 void modArr(int *arr, size_t s) {
04   for(int i=0; i<s; ++i) {
05     arr[i]*=2;
06   }
07 }
```

```
08 void printArr(const int *arr, size_t s) {
09   for(int i=0; i<s; ++i) {
10     printf("%d \n", arr[i]);
11   }
12   printf("\n");
13 }

14 int main(void) {
15   int iarr[] = { 11, 22, 33, 44, 55 };
16   modArr(iarr, sizeof(iarr)/sizeof(int));
17   printArr(iarr, sizeof(iarr)/sizeof(int));
18   return EXIT_SUCCESS;
19 }
```

In den Zeilen **(03)** bis **(07)** finden Sie die Funktion `modArr()`, die nichts anders tut, als den Inhalt der einzelnen Werte eines an die Funktion übergebenen Arrays zu verdoppeln. In der Zeile **(16)** rufen Sie diese Funktion mit der Anfangsadresse des Arrays und der Anzahl der darin enthaltenen Elemente auf. Diese wurde hier ganz einfach mit `sizeof(array)/sizeof(Datentyp)` übergeben. Ein reines `sizeof(array)` würde nur die Größe des Arrays zurückliefern, weshalb Sie diesen Wert noch durch die Größe des Datentyps teilen müssen, um die Anzahl der Elemente zu ermitteln.

Die zweite Funktion `printArr()` in den Zeilen **(08)** bis **(13)** hat einen `const`-Arrayparameter, um so ein versehentliches Ändern der Arrayelemente in der Funktion zu verhindern. Die Funktion gibt die Werte der einzelnen Arrayelemente aus und wird in der Zeile **(17)** mit der Anfangsadresse des Arrays und der Anzahl der Elemente aufgerufen.

10.9 char-Arrays und Zeiger

Was ich zu den Zeigern mit Arrays zuvor beschrieben habe, gilt natürlich auch in Bezug auf Zeiger mit Strings (alias `char`-Array). Nur gibt es zwischen dem `char`-Zeiger und dem `char`-Array einen kleinen Unterschied, den Sie unbedingt kennen müssen.

Bei einer Array-Deklaration

```
char str[] = "hallo";
```

wird ein Array mit sechs Elementen angelegt, um den String "hallo" zu speichern. Das letzte Element ist \0. Der Bezeichner str ist dabei eine **konstante Anfangsadresse** des Arrays und kann nicht geändert werden.

Deklarieren Sie hingegen eine Zeigerversion wie

```
char *str = "hallo";
```

wird ein extra Speicherplatz für den Zeiger verwendet. Hiermit legen Sie praktisch den Zeiger str an, der auf den Anfang, das Zeichen *h* im konstanten String "hallo", zeigt. Der String "hallo" selbst befindet sich irgendwo anders im Speicher. Des Weiteren ist es hier auch zeigertypisch möglich, die Adresse, auf die str verweist, zu ändern.

Ein Beispiel:

```
char *str1 = "hallo";
char str2[] = "welt";
printf("%s\n", str1);  // hallo
str1 = str2;           // zeigt jetzt auf den Anfang von str2
printf("%s\n", str1);  // welt
```

Hier wurde die Anfangsadresse des Zeigers str1 geändert. Sie zeigt jetzt auf die Anfangsadresse von str2. Wenn Sie allerdings den Zeiger str1 auf str2 verweisen lassen, ist der String "hallo", auf den der Zeiger str1 zuvor gezeigt hat, im Speicher zwar noch vorhanden; aber es gibt keine Möglichkeit mehr, auf ihn zuzugreifen, weil die Adresse nirgendwo gespeichert wurde.

10.10 Arrays von Zeigern

Arrays von Zeigern (auch: Zeigerarrays) können je nach Anwendung als Alternative für zweidimensionale Arrays eingesetzt werden, ganz besonders gerne bei zweidimensionalen char-Arrays (Strings):

```
char german[10][50] = {
     "eins", "zwei", "drei", "vier", "fünf",
     "sechs", "sieben", "acht", "neun", "zehn"
};

char english[10][50] = {
     "one", "two", "three", "four", "five",
     "six", "seven", "eight", "nine", "ten"
};
```

Sie sehen hier zwei zweidimensionale Arrays mit Zeichenketten, die jeweils 10 Elemente mit jeweils 50 Zeichen aufnehmen können. Hier fällt sofort die Speicherverschwendung auf, weil keiner der Strings 50 Zeichen benötigt. Der längste String in der Tabelle benötigt gerade mal sieben Zeichen. Und was ist, wenn Sie noch mehr Strings hinzufügen wollen? Klar, Sie können ein größeres zweidimensionales Array verwenden. Allerdings wird hierfür meistens wieder zu viel Speicher reserviert.

Ein zweidimensionales Array – wie hier mit mehreren Strings – hat also häufig das Problem, dass entweder unnötig viel Speicher belegt wird oder dass es unflexibel ist. Daher bieten sich für solche Fälle Arrays von Zeigern an. Dasselbe Beispiel mit solchen Arrays von Zeigern:

```
char *german[] = {
     "eins", "zwei", "drei", "vier", "fünf",
     "sechs", "sieben", "acht", "neun", "zehn"
};

char *english[] = {
     "one", "two", "three", "four", "five",
     "six", "seven", "eight", "nine", "ten"
};
```

Beide Arrays von Zeigern erfüllen hier denselben Zweck wie im vorigen Beispiel. Sie haben allerdings den Vorteil, dass jetzt nur so viel Speicherplatz wie nötig verwendet wird.

Ein solches Array speichert nicht mehr die Strings selbst, sondern Zeiger auf diese Strings, die sich irgendwo auf dem Speicher befinden können.

Benötigen Sie ein Array mit einer bestimmten Anzahl von Zeigern eines Datentyps, können Sie dies folgendermaßen deklarieren:

```
char *strings[100];
```

Hiermit haben Sie 100 Zeiger auf Strings angelegt. Richtig sinnvoll können Sie ein Array mit solchen Zeigern allerdings erst verwenden, wenn Sie Kenntnisse der dynamischen Speicherverwaltung besitzen. Die einzelnen Adressen können Sie mithilfe des Indizierungsoperators vergeben. Hier ein Beispiel:

```
strings[0] = "Hallo";   // 1. Zeiger auf Stringkonstante
strings[1] = "Welt";    // 2. Zeiger auf Stringkonstante
...
```

Hierzu noch ein kurzes Listing, das die einfache Verwendung von Arrays von Zeigern in der Praxis demonstrieren soll:

```
00  // kap010/listing008.c
01  #include <stdio.h>
02  #include <stdlib.h>

03  int main(void) {
04    char *german[] = {
       "eins", "zwei", "drei", "vier", "fuenf",
       "sechs", "sieben", "acht", "neun", "zehn"
      };
05    char *english[] = {
        "one", "two", "three", "four", "five",
        "six", "seven", "eight", "nine", "ten"
      };
06    int ival = 0;
07    printf("Bitte eine Ganzzahl von 1 bis 10: ");
08    if( scanf("%d", &ival) != 1 ) {
09      printf("Fehler bei der Eingabe\n");
```

```
10      return EXIT_FAILURE;
11    }
12    else if( ival <= 0 || ival > 10 ) {
13      printf("Dies ist kein Wert von 1 bis 10\n");
14      return EXIT_FAILURE;
15    }
16    else {
17      printf("Ganzzahl: %d\n", ival);
18      printf("Englisch: %s\n", english[ival-1]);
19      printf("Deutsch : %s\n", german[ival-1]);
20    }
21    return EXIT_SUCCESS;
22  }
```

Im Beispiel werden Sie nach einer Ganzzahl von 1 bis 10 gefragt. Je nach Eingabe der Ganzzahl erhalten Sie in den Zeilen **(18)** und **(19)** den entsprechenden ganzzahligen Wert als englisches und deutsches Wort.

Das Programm bei der Ausführung:

```
Bitte eine Ganzzahl von 1 bis 10: 8
Ganzzahl: 8
Englisch: eight
Deutsch : acht
```

10.11 void-Zeiger

Ein Zeiger auf `void` ist ein nicht typisierter und vielseitiger Zeiger, der kompatibel mit allen anderen Zeigertypen ist und bei der Zuweisung mit typisierten Zeigern vermischt werden kann. Dabei darf der `void`-Zeiger an einen typisierten Zeiger zugewiesen werden, und genauso darf an den `void`-Zeiger ein typisierter Zeiger zugwiesen werden. Mit einem `void`-Zeiger umgehen Sie praktisch die Typüberprüfung des Compilers. In der Regel ist es ja sonst nicht erlaubt, ohne eine explizite Typenkonvertierung Zeiger eines Datentyps an einen Zeiger eines anderen Datentyps zuzuweisen.

Einschränkungen von »void«-Zeigern

Der `void`-Zeiger selbst kann nicht dereferenziert werden. Sie können den `void`-Zeiger nicht für den Zugriff auf Objekte verwenden. Mit dem `void`-Zeiger sind nur folgende Operationen möglich:

- Vergleich von `void`-Zeigern mit anderen Zeigern
- Umwandlung von `void`-Zeigern in gültige Zeigertypen
- Zuweisung von Adressen an `void`-Zeiger

Wenn der Datentyp des Zeigers noch nicht feststeht, wird der `void`-Zeiger verwendet. `void`-Zeiger haben den Vorteil, dass Sie diesen eine beliebige Adresse zuweisen können. Auch viele Funktionen der Standardbibliothek haben einen `void`-Zeiger im Prototyp definiert. Bestes Beispiel für einen solchen Prototypen ist die Funktion `malloc()` zum dynamischen Reservieren von Speicher. Hier die Syntax dazu:

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

Der Rückgabetyp `void*` ist sinnvoll, weil vor der Verwendung von `malloc()` noch nicht bekannt ist, von welchem Zeigertyp der Speicher reserviert wird. Bei der Verwendung wird der `void`-Zeiger in den entsprechenden Zeigertyp, dem er zugewiesen wird, umgewandelt. Hier ein Beispiel:

```
// Speicher für 100 Elemente vom Typ float reserviert
float* fval = malloc( 100 * sizeof(float) );
// Speicher für 255 Elemente vom Typ int reserviert
int* ival = malloc( 255 * sizeof(int) );
```

Hat eine Funktion einen `void`-Zeiger als formalen Parameter, wird beim Funktionsargument der Typ in `void*` umgewandelt. So macht es beispielsweise die Funktion `memcmp()` aus der Headerdatei `<string.h>`:

```
int memcmp(const void* v1, const void* v2, size_t n);
```

Anhand dieser beiden Standardfunktionen können Sie auch den Vorteil der `void`-Zeiger erkennen. Anstatt für jeden Datentyp eine Funktion zu schreiben, wird einfach eine Version mit einem typenlosen `void`-Zeiger erstellt.

10.12 Typ-Qualifizierer bei Zeigern

Bei der Deklaration von Zeigern können Sie die Typ-Qualifizierer `const`, `volatile` und `restrict` verwenden. Hier sollen einige gängige Anwendungen der Typ-Qualifizierer in Bezug auf die Zeiger beschrieben werden.

10.12.1 Konstanter Zeiger

Da die zugewiesenen Adressen von konstanten Zeigern zur Laufzeit nicht mehr versetzt werden können, müssen diese schon bei der Definition mit einer Adresse initialisiert werden. Wohlgemerkt: Das Speicherobjekt, auf das der konstante Zeiger verweist, ist nicht konstant. Folgender Codeausschnitt soll dies demonstrieren:

```
01  int iarr[] = { 11, 22, 33 };  // int-Array
02  int* const c_ptr = &iarr[1];  // konstanter Zeiger
03  *c_ptr = 44;                  // iarr[1] erhält neuen Wert
04  c_ptr++;                      // Fehler!!!
05  *c_ptr = 66;
```

In der Zeile **(02)** weisen Sie dem konstanten Zeiger die Adresse vom zweiten Element des `iarr`-Arrays zu. Diese gespeicherte Adresse in `c_ptr` kann jetzt nicht mehr geändert werden. In der Zeile **(03)** ändern Sie den Inhalt des zweiten Elements in `iarr` auf den Wert 44. In der Zeile **(04)** wird versucht, den Zeiger auf das nächste Element im Array (hier `iarr[2]`) zu setzen, um diesen Wert dann in der Zeile **(05)** zu ändern. Allerdings wird sich das Beispiel ab der Zeile **(04)** nicht mehr übersetzen lassen, weil hier versucht wird, die zugewiesene Adresse des konstanten Zeigers zu ändern, sodass es zur Zeile **(05)** gar nicht mehr kommt. `c_ptr` ist hier ein konstanter Zeiger auf ein `int`.

10.12.2 Zeiger für konstante Daten

Benötigen Sie einen reinen Zeiger nur zum Lesen des Inhalts, auf den er verweist, müssen Sie einen Zeiger auf `const` verwenden. Damit ist nur noch das Lesen mit dem Indirektionsoperator auf das referenzierte Speicherobjekt möglich. Das Speicherobjekt, auf das der Zeiger verweist, muss hingegen nicht konstant sein. Ein kurzer Codeausschnitt soll einen solchen Zeiger demonstrieren:

```
01  int iarr[] = { 11, 22, 33 };  // int-Array
02  int const * c_ptr= &iarr[0];  // Zeiger zum Lesen auf Array
03  *c_ptr = 44;                  // Fehler!!!
04  // Folgendes ist erlaubt, da nur gelesen wird:
05  for(size_t i=0; i<sizeof(iarr)/sizeof(int); i++, c_ptr++){
06    printf("%d\n", *c_ptr);
07  }
```

In der Zeile **(02)** legen Sie einen Zeiger auf kontante Arrayelemente fest. Sie können diesen Zeiger nicht verwenden, um den Inhalt der Arrayelemente zu ändern (wie es in der Zeile **(03)** versucht wird). Ein lesender Zugriff wie in den Zeilen **(05)** bis **(07)** ist hingegen jederzeit möglich. `c_ptr` ist hier ein Zeiger auf ein konstantes `int`.

10.12.3 Konstanter Zeiger und Zeiger für konstante Daten

Neben den beiden eben erwähnten Möglichkeiten gibt es noch eine dritte Möglichkeit, bei der Sie weder die Adresse noch den Wert, auf den der Zeiger verweist, ändern können. Diese Form sieht folgendermaßen aus:

```
01  int iarr[] = { 11, 22, 33 };  // int-Array
02  int const * const c_ptr = &iarr[1];
03  *c_ptr = 44;                  // Fehler!!!
04  c_ptr++;                      // Fehler!!!
05  printf("%d\n", *c_ptr);       // OK, da lesend
```

Durch die Verwendung von `const` vor und nach dem Zeiger verweist der Zeiger auf das zweite Element im Array `iarr`. Bei einem solchen Zeiger können Sie weder den Inhalt (wie in der Zeile **(03)** zu sehen), noch die Adresse, auf die der Zeiger verweist (siehe Zeile **(04)**) ändern. Ein lesender Zugriff (wie in der Zeile **(05)** zu sehen) ist nach wie vor möglich. Jetzt ist `c_ptr` ein konstanter Zeiger auf einen konstanten `int`.

10.12.4 Konstante Parameter für Funktionen

Der Qualifizierer `const` wird sehr gerne in Funktionen bei den formalen Parametern verwendet. Auch viele Standardfunktionen machen regen

Gebrauch davon. Betrachten Sie beispielsweise die Syntax der Funktion `printf()`:

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

Dank des Zeigers auf `const` ist es ausgeschlossen, dass innerhalb der Ausführung der Funktion `printf()` ein schreibender Zugriff auf `format` vorgenommen werden kann. Auf den Zeiger innerhalb der Funktion kann somit nur lesend zugegriffen werden.

10.12.5 restrict-Zeiger

Mit dem C99-Standard wurde der Typ-Qualifizierer `restrict` neu eingeführt. Dieses Schlüsselwort qualifiziert sogenannte `restrict`-Zeiger. Der `restrict`-Zeiger hat eine enge Beziehung zu dem Speicherobjekt, auf das er verweist. Ein Beispiel:

```
int* restrict iRptr = malloc (10 * sizeof (int) );
```

Damit schlagen Sie vor, dass Sie den von `malloc()` zurückgegebenen reservierten Speicher nur mit dem Zeiger `iRptr` verwenden. Wohlgemerkt: Mit dem Qualifizierer `restrict` geben Sie dem Compiler das Versprechen, dass Sie auf das Speicherobjekt ausschließlich mit diesem Zeiger zurückgreifen. Jede Manipulation außerhalb des `restrict`-Zeigers, und sei es nur lesend, ist unzulässig.

Es ist Ihre Aufgabe zu überprüfen, ob der `restrict`-Zeiger richtig verwendet wird und Sie nur über diesen Zeiger auf ein Speicherobjekt zugreifen. Der Compiler kann nicht überprüfen, ob Sie Ihr Versprechen eingehalten haben. Falls Sie die Regeln nicht einhalten, gibt es zwar keine Fehlermeldung des Compilers und häufig auch keine Probleme bei der Ausführung des Programms, aber dennoch ist das Verhalten laut Standard undefiniert.

Der Vorteil des `restrict`-Zeigers ist, dass Sie es dem Compiler ermöglichen, Optimierungen des Maschinencodes durchzuführen. Allerdings muss der Compiler diesem Hinweis nicht nachkommen und kann den Qualifizierer `restrict` auch ignorieren. Der `restrict`-Zeiger kann auch sehr gut bei Funktionen verwendet werden. Er zeigt dann an, dass sich zwei

Zeiger in der Parameterliste nicht überlappen, sprich dasselbe Speicherobjekt verwenden dürfen. Beispielsweise ist bei

```
int cpy_number( int* v1, int* v2 ) {
/* ... */
}
```

nicht klar angegeben, ob sich die beiden Speicherobjekte, auf die die Zeiger `v1` und `v2` verweisen, überlappen dürfen oder nicht. Mit dem neuen Qualifizierer `restrict` ist dies jetzt sofort erkennbar:

```
int cpy_number( int* restrict v1, int* restrict v2 ) {
/* ... */
}
```

Wird trotzdem versucht, die Funktion mit sich überlappenden Speicherobjekten aufzurufen, ist das weitere Verhalten undefiniert. Ein ungültiger Aufruf kann beispielsweise wie folgt aussehen:

```
// Unzulässig wegen der restrict-Zeiger,
// zwei gleiche Speicherobjekte werden verwendet,
// die sich somit überlappen
val = cpy_number( &x, &x );
// OK, zwei verschiedene Speicherobjekte
val = cpy_number( &x, &y );
```

Vom `restrict`-Zeiger wird seit C99 rege in der Standardbibliothek Gebrauch gemacht. Beispielsweise sieht die Syntax der Funktion `strncpy()`aus der Headerdatei `<string.h>` wie folgt aus:

```
#include <string.h>
char* strncpy(
          char* restrict s1,
          const char* restrict s2,
          size_t n );
```

Die Funktion kopiert `n` Bytes vom Quell-Array `s2` in das Ziel-Array `s1`. Da die beiden Zeiger als `restrict` deklariert sind, müssen Sie beim Aufruf der

Funktion beachten, dass die Zeiger nicht auf dieselben Speicherobjekte verweisen, sich also nicht überlappen. Betrachten Sie dazu folgendes Beispiel:

```
char arr1[20];
char arr2[] = { "Hallo Welt" };
// Ok, 10 Zeichen von arr2 nach arr1 kopieren
strncpy( arr1, arr2, 10 );
arr1[10] = '\0'
// Unzulässig, Speicherbereiche überlappen sich,
// das gibt nur Datensalat
strncpy( arr1, arr1, 5 );
```

10.13 Zeiger auf Funktionen

Zeiger können auch auf Funktionen verweisen. Sie verweisen auf die Anfangsadresse des Codes der Funktion. Wie bei der Deklaration eines Zeigers auf ein Array sind hierbei zusätzliche Klammern nötig. Ein solcher Zeiger kann wie folgt erstellt werden:

```
Rückgabetyp (*funktionsZeiger)(formale Parameter);
```

Sie haben bei dieser Deklaration einen Zeiger auf einen Funktionstyp mit einem `int`- oder mehreren formalen Parametern und einem Rückgabetyp deklariert. Ohne die Klammerungen von `(*funktionsZeiger)` hätten Sie lediglich den Prototypen einer Funktion und keine Definition eines Zeigers erstellt. Der Name der Funktion wird dann implizit in einen Zeiger auf diese Funktion umgewandelt. Aufrufen können Sie die Funktion über Zeiger entweder mit

```
( *funktionsZeiger ) (parameter);
```

womit der Funktionszeiger explizit dereferenziert wird, oder auch einfach mit

```
funktionsZeiger(parameter);
```

womit der Funktionszeiger – ähnlich wie beim C-Arraynamen – automatisch dereferenziert wird.

Folgendermaßen können Sie beispielsweise einen Zeiger auf die Standardfunktion `printf()` erstellen und verwenden:

```
00  // kap010/listing009.c
01  #include <stdio.h>
02  #include <stdlib.h>

03  int main(void) {
04    int (*fptr)(const char*, ...);
05    fptr = printf;
06    (*fptr)("Hallo Welt mit Funktionszeigern\n");
07    return EXIT_SUCCESS;
08  }
```

In Zeile **(04)** erstellen Sie einen Zeiger, der auf die Funktion `printf()` der Standardbibliothek verweisen kann. Natürlich müssen Sie bei einem solchen Funktionszeiger darauf achten, dass der Rückgabewert und die Parameter mit dem Prototyp der Funktion übereinstimmen, auf die Sie verweisen wollen:

```
// Prototyp: printf
int printf(const char*, ...);
// Unser Zeiger auf die Funktion lautet daher:
int (*fptr)(const char*, ...);
```

Das bedeutet auch, dass Sie mit diesem Zeiger nicht nur auf `printf()` verweisen können, sondern auch auf alle anderen Funktionen mit demselben Rückgabewert und Funktionsparameter. Im Beispiel könnten Sie daher auch die Funktion `scanf()` oder eine eigene Funktion mit denselben Rückgabe- und Parameterangaben verwenden.

In Zeile **(05)** bekommt der Zeiger `fptr` die Adresse für die Funktion `printf` zugewiesen. Diese kann jetzt, wie in Zeile **(06)** geschehen, über den Zeiger aufgerufen werden. Alternativ könnten Sie den Aufruf der Zeile **(06)** auch wie folgt notieren:

```
06    fptr("Hallo Welt mit Funktionszeigern\n");
```

Zeiger auf Funktionen können aber auch in einem Array gespeichert werden. Dann können die einzelnen Funktionen anschließend über den Index aufgerufen werden. Ähnliches könnte beispielsweise bei einem Tastaturtreiber verwendet werden, bei dem je nach Tastendruck in eine entsprechende Funktion verzweigt wird. Zeiger auf Funktionen mit Arrays sehen folgendermaßen aus:

```
int (*funktionsZeiger[])(int parameter);
```

Den Klammern wurde lediglich noch der Indizierungsoperator [] hinzugefügt. Hierzu ein sehr einfaches Rechenbeispiel, das Zeiger auf eine Funktion mit Arrays demonstriert:

```
00 // kap010/listing010.c
01 #include <stdio.h>
02 #include <stdlib.h>

03 double addition( double x, double y ) {
04   return (x + y);
05 }

06 double subtraktion( double x, double y ) {
07   return (x - y);
08 }

09 double multiplikation( double x, double y ) {
10   return (x * y);
11 }

12 double division( double x, double y ) {
13   return (x / y);
14 }

15 double (*fptr[4])(double d1, double d2) = {
     addition, subtraktion, multiplikation, division
   };
```

```
16  int main(void) {
17    double v1=0.0, v2=0.0;
18    int operator=-1;
19    printf("Zahl1 :-> ");
20    if( scanf("%lf", &v1) != 1 ) {
21      printf("Fehler bei der Eingabe\n");
22      return EXIT_FAILURE;
23    }
24    printf("Zahl2 :-> ");
25    if( scanf("%lf", &v2)  != 1 ) {
26      printf("Fehler bei der Eingabe\n");
27      return EXIT_FAILURE;
28    }
29    printf("Welcher Operator soll verwendet werden?\n");
30    printf("0 = +\n1 = -\n2 = *\n3 = /\n");
31    printf("Ihre Auswahl: ");
32    if( scanf("%d", &operator)  != 1 ) {
33      printf("Fehler bei der Eingabe\n");
34      return EXIT_FAILURE;
35    }
36    if( ! (operator >= 0 && operator <= 3) ) {
37      printf("Fehler beim Operator\n");
38      return EXIT_FAILURE;
39    }
40    printf("Ergebnis: %lf\n", fptr[operator](v1, v2));
41    return EXIT_SUCCESS;
42  }
```

In diesem Beispiel werden zwei `double`-Zahlen eingelesen, und damit wird eine einfache Berechnung durchgeführt. Die mathematischen Funktionen werden über Zeiger aufgerufen, die im Array `fptr` stehen und in Zeile **(15)** deklariert wurden. In Zeile **(40)** wird die Funktion aufgerufen, auf die `fptr[operator]` verweist.

Zusammengefasst können solche Funktionszeiger an Funktionen übergeben, von Funktionen zurückgegeben, in Arrays gespeichert und anderen Funktionszeigern zugewiesen werden.

10.14 Kontrollfragen und Aufgaben

1. Versuchen Sie, die grundlegende Funktion von Zeigern zu erläutern. Woran erkennt man einen Zeiger?
2. Welche Gefahr entsteht beim Initialisieren von Zeigern?
3. Was versteht man unter einer Dereferenzierung?
4. Was ist der `NULL`-Zeiger, und wozu ist er gut?
5. Was ist ein `void`-Zeiger, und wozu wird er verwendet?
6. Da die Position der Typ-Qualifizierer recht flexibel ist, ergeben sich unterschiedliche Möglichkeiten, bei denen besonders gerne der Qualifizierer `const` verwendet wird. Wenn `const` bei Zeigern verwendet wird – ab wann ist dann die Rede von konstanten Daten und ab wann von konstanten Zeigern?
7. Welcher Fehler wurde hier gemacht?

```
01  int *ptr;
02  int ival;
03  ptr = ival;
04 *ptr = 255;
```

8. Welcher Wert wird mit den beiden `printf`-Anweisungen in den Zeilen **(06)** und **(07)** ausgegeben?

```
01  int *ptr = NULL;
02  int ival;
03  ptr = &ival;
04  ival = 98765432;
05  *ptr = 12345679;
06  printf("%d\n", ival);
07  printf("%d\n", *ptr);
```

9. Was geben die `printf`-Anweisungen in den Zeilen **(07)** bis **(11)** auf dem Bildschirm aus?

```
01  int iarray[] = { 12, 34, 56, 78, 90, 23, 45 };
02  int *ptr1=NULL, *ptr2=NULL;

03  ptr1 = iarray;
04  ptr2 = &iarray[4];
```

```
05  ptr1+=2;
06  ptr2++;

07  printf("%d\n", *ptr1);
08  printf("%d\n", *ptr2);
09  printf("%td\n", ptr2 - ptr1);
10  printf("%d\n", (ptr1 < ptr2));
11  printf("%d\n", ((*ptr1) < (*ptr2)));
```

10. Das folgende Listing soll in einem String einen bestimmten Buchstaben suchen und ab dieser Fundstelle eine Adresse auf dem String zurückgeben. Im Beispiel wird mit dem Aufruf der Funktion `mysearch()` in der Zeile **(18)** »Hallo Welt« nach dem Buchstaben W gesucht. Die Funktion gibt nur noch einen String bzw. die Anfangsadresse ab W zurück. Im Beispiel müsste die `printf`-Ausgabe der Zeile **(19)** somit »Welt« lauten. Im Beispiel werden allerdings nur scheinbar beliebige Zeichen ausgegeben. Wo liegt der Fehler?

```
00  // kap010/aufgabe001.c
01  #include <stdio.h>
02  #include <stdlib.h>
03  #include <string.h>
04  #define MAX 255

05  char *mysearch( char *str, char ch ) {
06    char buf[MAX] = "";
07    int i = 0;
08    while( str[i] != '\0' ) {
09      if( str[i] == ch ) {
10        strncpy(buf, &str[i], MAX);
11        return buf;
12      }
13      i++;
14    }
15    return buf;
16  }

17  int main(void) {
```

```
18    char *str = mysearch("Hallo Welt", 'W');
19    printf("%s\n", str);
20    return EXIT_SUCCESS;
21  }
```

11. Wo ist der Unterschied zwischen den folgenden beiden Deklarationen?

```
char *str1  = "Hallo Welt";
char str2[] = "Hallo Welt";
```

12. Schreiben Sie eine Funktion, mit der Sie einen Ganzzahlenwert (`int`) in einen String konvertieren können.

13. Verbessern Sie im folgenden Listing die Funktion `Vquader()`, mit der das Volumen eines Quaders berechnet und zurückgegeben wird. Verwenden Sie für das Programm Zeiger als Parameter anstatt die Übergabe als Kopie.

```
01  double Vquader(double a, double b, double c) {
02    static double volumen;
03    volumen = a * b * c;
04    return volumen;
05  }
```

14. Schreiben Sie ein Programm, das einen String von der Tastatur einliest und diesen anhand der Trennzeichen `" ,.:!?\t\n"` in einzelne Wörter zerlegt. Speichern Sie die einzelnen Wörter in Zeigerarrays. **Tipp:** Sie können zum Zerlegen auch die Funktion `strtok()` aus der Headerdatei `<string.h>` verwenden.

Kapitel 11

Dynamische Speicherverwaltung

Bisher kennen Sie nur die Möglichkeit, einen statischen Speicher in Ihren Programmen zu verwenden. In der Praxis kommt es allerdings häufig vor, dass zur Laufzeit des Programms gar nicht bekannt ist, wie viele Daten gespeichert werden und wie viel Speicher somit benötigt wird. Natürlich können Sie immer ein Array mit besonders viel Speicher anlegen. Allerdings müssen Sie hierbei Folgendes beachten:

- Auch ein Array mit extra viel Speicherplatz ist nicht unendlich und mit einer statischen Speichergröße beschränkt. Wird die maximale Größe erreicht, stehen Sie wieder vor dem Problem Speicherknappheit. Zwar gäbe es hier VLA-Arrays, aber diese müssen seit C11 nur noch optional vorhanden sein.
- Verwenden Sie ein Array mit extra viel Speicherplatz, verschwenden Sie sehr viele Ressourcen, die vielleicht von anderen Programmen auf dem System dringender benötigt werden.
- Die Gültigkeit des Speicherbereichs eines Arrays verfällt, sobald der Anweisungsblock verlassen wurde. Ausnahmen stellen globale und als `static` deklarierte Arrays dar.

In vielen Fällen dürfte es daher sinnvoller sein, den Speicher erst zur Laufzeit des Programms dynamisch zu reservieren, sobald er benötigt wird, bzw. ihn wieder freizugeben, wenn er nicht mehr benötigt wird. Realisiert wird die dynamische Speicherverwaltung mit den Zeigern und einigen Funktionen der C-Standardbibliothek.

Im Gegensatz zu einem statischen Speicherbereich wie beispielsweise einem Array bedeutet das, dass hierbei ein gewisser Mehraufwand betrieben werden muss. Die Freiheit in C, Speicher mithilfe von Zeigern zu reservieren und zu verwalten, birgt natürlich auch Gefahren. Sie könnten beispielsweise auf eine undefinierte Adresse im Speicher zugreifen. Sie

müssen sich auch darum kümmern, Speicher, den Sie reserviert haben, wieder freizugeben. Tun Sie das nicht, entstehen sogenannte Speicherlecks (engl. *memory leaks*), die bei länger oder dauerhaft laufenden Programmen (beispielsweise Serveranwendungen) den Speicher füllen. Irgendwann kommen Sie dann unter Umständen um einen Neustart des Programms nicht mehr herum.

Der dynamische Speicherbereich, in dem Sie zur Laufzeit des Programms etwas anfordern können, wird **Heap** (oder auch Freispeicher) genannt. Wenn Sie Speicher von diesem Heap anfordern, erhalten Sie immer einen zusammenhängenden Bereich und nie einzelne Fragmente. Einen weiteren Speicherbereich mit einem statischen und fest reservierten Speicherplatz haben Sie mit dem Stack bereits in Abschnitt 7.6, »Exkurs: Funktion bei der Ausführung«, kennengelernt.

11.1 Neuen Speicherblock reservieren

Für die einfache Anforderung von Speicher finden Sie in der Headerdatei `<stdlib.h>` die Funktionen `malloc()` und `calloc()`. Zuerst die Syntax von `malloc()`:

```
#include <stdlib.h>
void* malloc(size_t size);
```

Die Funktion `malloc()` fordert nur so viel zusammenhängenden Speicher an, wie im Parameter `size` angegeben wurde. Der Parameter `size` verwendet Bytes als Einheit. Zurückgegeben wird ein typenloser `void`-Zeiger mit der Anfangsadresse des zugeteilten Speicherblocks. Konnte kein zusammenhängender Speicherblock angefordert werden, wie er mit `size` angegeben wurde, liefert die Funktion `NULL` zurück. Der Inhalt des reservierten Speicherbereichs ist am Anfang undefiniert.

Benötigen Sie Speicher für 100 Variablen vom Typ `int`, können Sie diesen Speicherblock folgendermaßen reservieren:

```
int *iptr;
iptr=malloc(100 * sizeof(int)); // Speicher für 100 int-Typen
```

Hier lassen Sie bei der Reservierung von Speicher immer den `sizeof`-Operator entscheiden, wie groß der zu reservierende Datentyp ist.

Es ist auch möglich, direkt die Größe des zu reservierenden Speichers wie folgt anzufordern – bezogen auf das eben gezeigte Beispiel könnten Sie ebenfalls Speicher für 100 `int`-Typen reservieren:

```
int* iptr;
iptr = malloc (400);   // 400 Bytes Speicher reservieren
```

Diese Art der Speicherreservierung ist allerdings mit Vorsicht zu genießen. Sie setzen in diesem Beispiel voraus, dass `int` auf dem System, auf dem das Programm übersetzt wird, vier Bytes breit ist. Das mag vielleicht relativ häufig zutreffen. Was aber, wenn ein `int` auf einem anderen System unterschiedlich breit ist? Sicherer ist es daher, den `sizeof`-Operator dafür zu verwenden.

Hier ein Listing, wie Sie einen beliebigen Speicher dynamisch zur Laufzeit reservieren können:

```
00  // kap011/listing001.c
01  #include <stdio.h>
02  #include <stdlib.h>

03  int* iArray( unsigned int n ) {
04    int* iptr = malloc( n *(sizeof(int) ) );
05    if( iptr != NULL ) {
06      for(unsigned int i=0; i < n; i++) {
07        iptr[i] = i*i;  // Alternativ: *(iptr+i)=...
08      }
09    }
10    return iptr;
11  }

12  int main(void) {
13    unsigned int val=0;
14    printf("Wie viele Elemente benoetigen Sie: ");
15    if( scanf("%u", &val) != 1 ) {
```

```
16      printf("Fehler bei der Eingabe\n");
17      return EXIT_FAILURE;
18    }
19    int* arr = iArray( val );
20    if( arr == NULL ) {
21      printf("Fehler bei der Speicherreservierung!\n");
22      return EXIT_FAILURE;
23    }
24    printf("Ausgabe der Elemente\n");
25    for(unsigned i=0; i < val; i++ ) {
26      printf("arr[%u] = %u\n", i, arr[i]);
27    }
28    if(arr != NULL) {
29      free(arr);  // Freigabe des Speichers
30    }
31    return EXIT_SUCCESS;
32  }
```

Im Beispiel werden Sie gefragt, für wie viele `int`-Elemente Sie einen Speicherplatz reservieren wollen. In Zeile **(19)** rufen Sie dann die Funktion `iArray()` mit der Anzahl der gewünschten `int`-Elemente auf. Die Adresse des Rückgabewerts übergeben Sie dem Zeiger `arr`. In der Funktion `iArray()` reservieren Sie in Zeile **(04)** `n` Elemente vom Typ `int`. In Zeile **(05)** wird überprüft, ob die Reservierung nicht `NULL` ist und ob Sie erfolgreich Speicher vom System erhalten haben. Ist dies der Fall, übergeben Sie den einzelnen Elementen in der `for`-Schleife (Zeile **(06)** bis **(08)**) beliebige Werte.

Der Zugriff auf die einzelnen Elemente mit `iptr[i]` oder `*(iptr+i)` wurde bereits in Kapitel 10, »Zeiger (Pointer)«, beschrieben und sollte kein Problem mehr darstellen. Nebenbei erwähnt: Sie haben in diesem Listing auch gleichzeitig ein **dynamisches Array** erstellt und verwendet.

Am Ende der Funktion, in der Zeile **(10)**, geben Sie die Anfangsadresse auf den reservierten Speicherblock an den Aufrufer der Zeile **(19)** zurück. An dieser Stelle werden Sie festgestellt haben, dass Sie reservierten Speicher problemlos vom Heap aus an Funktionen zurückgeben können.

In der `main`-Funktion wird in Zeile **(20)** noch überprüft, ob der Rückgabewert `NULL` war. Würde dies zutreffen, wäre die Speicherreservierung in der Zeile **(04)** fehlgeschlagen. In der `for`-Schleife (Zeile **(25)** bis **(27)**) werden die einzelnen Elemente ausgegeben.

Auf die Freigabe des Speichers mit der Funktion `free()`, die in der Zeile **(29)** verwendet wurde, wird noch in Abschnitt 11.3, »Speicherblock freigeben«, gesondert eingegangen. Auf die Überprüfung in der Zeile **(28)**, ob `arr` ungleich `NULL` ist, um dann den Speicher freizugeben, könnten Sie auch verzichten, weil auch ein `free(NULL)` erlaubt ist. Aber mehr dazu in Kürze.

Das Programm bei der Ausführung:

```
Wie viele int-Elemente benoetigen Sie: 9
Ausgabe der Elemente
arr[0] = 0
arr[1] = 1
arr[2] = 4
arr[3] = 9
arr[4] = 16
arr[5] = 25
arr[6] = 36
arr[7] = 49
arr[8] = 64
```

Benötigen Sie eine Funktion, die neben der Reservierung eines zusammenhängenden Speichers auch noch den zugeteilten Speicher automatisch mit 0 initialisiert, können Sie die Funktion `calloc()` verwenden. Die Syntax unterscheidet sich geringfügig von der von `malloc()`:

```
#include <stdlib.h>
void* calloc( size_t count, size_t size );
```

Hiermit reservieren Sie `count` × `size` Bytes zusammenhängenden Speicher. Es wird ein typenloser Zeiger auf `void` mit der Anfangsadresse des zugeteilten Speicherblocks zurückgegeben. Jedes Byte des reservierten Speichers wird außerdem automatisch mit 0 initialisiert. Konnte kein zusammen-

hängender Speicher mit `count × size` Bytes reserviert werden, gibt diese Funktion `NULL` zurück.

»malloc()« versus »calloc()«

Der Vorteil von `calloc()` gegenüber `malloc()` liegt darin, dass `calloc()` jedes Byte mit 0 initialisiert. Allerdings bedeutet das auch, dass `calloc()` mehr Zeit als `malloc()` beansprucht.

Hierzu ein kurzes Beispiel, wie Sie `calloc()` in der Praxis verwenden können:

```
00 // kap011/listing002.c
01 #include <stdio.h>
02 #include <stdlib.h>

03 int main(void) {
04   unsigned int val=0;
05   printf("Wie viele int-Elemente benoetigen Sie: ");
06   if( scanf("%u", &val) != 1 ) {
07     printf("Fehler bei der Eingabe\n");
08     return EXIT_FAILURE;
09   }
10   int* arr = calloc( val, (sizeof(int) ) );
11   if( arr == NULL ) {
12     printf("Fehler bei der Speicherreservierung!\n");
13     return EXIT_FAILURE;
14   }
15   printf("Ausgabe der Elemente\n");
16   for(unsigned int i=0; i < val; i++ ) {
17     printf("arr[%u] = %u\n", i, arr[i]);
18   }
19   if( arr != NULL ) {
20     free(arr);
21   }
22   return EXIT_SUCCESS;
23 }
```

Das Listing entspricht zum Teil dem Beispiel *listing001.c* zuvor. Hier wurde aber der Speicher mit `calloc()` in Zeile **(10)** in der `main`-Funktion reserviert und nicht mit einem Wert initialisiert. Dass `calloc()` alle Elemente mit 0 initialisiert, bestätigt die Ausgabe der `for`-Schleife in den Zeilen **(16)** bis **(18)**.

Die Funktion »aligned_alloc()«

Seit dem C11-Standard gibt es die Funktion `align_alloc()`, mit der Sie im Gegensatz zu `malloc()` neben der gewünschten Größe des Speicherblocks auch die Anordnung (*alignment*) im Speicher angeben können. Der Prototyp zu `align_alloc()` siehe wie folgt aus:

```
void* aligned_alloc(size_t alignment, size_t size);
```

Dies soll allerdings nur zu Ergänzung des Kapitels dienen. Auf das Anordnen von Speicherbereichen wird in diesem Buch nicht eingegangen.

11.2 Speicherblock vergrößern oder verkleinern

Mit der Funktion `realloc()` können Sie den Speicherplatz eines bereits zugeteilten Blocks vergrößern oder verkleinern. Hier die Syntax von `realloc()`:

```
#include <stdlib.h>
void* realloc( void* ptr, size_t size );
```

Mit dieser Funktion wird die Größe des durch `ptr` adressierten Speicherblock geändert, und es wird ein Zeiger auf die Anfangsadresse des neu veränderten Speicherblocks mit der Größe `size` Bytes zurückgegeben. Der Inhalt des ursprünglichen Speicherblocks bleibt erhalten, und der neue Speicherblock wird hinten angefügt. Ist das nicht möglich, kopiert `realloc()` den kompletten Inhalt in einen neuen Speicherblock. Damit ist weiterhin garantiert, dass ein zusammenhängender Speicherbereich reserviert wird.

Speicher um Blöcke erweitern

Wenn `realloc()` nach einem vorhandenen Speicherblock keinen zusammenhängenden Speicher mehr bekommt, um den Speicher zu vergrößern, muss der komplette vorhandene Speicherblock in einen zusammenhängenden Bereich umkopiert werden. Da das Umkopieren je nach Umfang der Daten ziemlich aufwendig werden kann, sollten Sie es vermeiden, `realloc()` für einzelne Elemente zu verwenden. Wenn es möglich ist, reservieren Sie mit `realloc()` immer größere Speicherblöcke für mehrere Elemente. Damit werden die `realloc()`-Aufrufe im Programm reduziert, was wiederum den Aufwand des Programms verringert und die Performance erhöht.

Wird für `ptr` ein `NULL`-Zeiger verwendet, funktioniert die Funktion `realloc()` wie `malloc()` und reserviert einen neuen Speicherblock mit `size` Bytes. Folgende Aufrufe sind somit identisch:

```
ptr = malloc( 100 * sizeof(int));
ptr = realloc( NULL, 100 * sizeof(int));
```

Kann `realloc()` keinen neuen Speicherplatz reservieren, wird auch hier der `NULL`-Zeiger zurückgegeben. Der ursprüngliche Speicherblock bleibt unverändert erhalten.

Verkleinern Sie den Speicherbereich, indem Sie für `size` eine kleinere Größe angeben, als der ursprüngliche Speicherblock groß war, wird der hintere Teil des Speicherblocks ab `size` freigegeben, und der vordere Teil bleibt erhalten.

```
// Speicher für 100 int-Elemente reserviert
ptr = malloc( 100 * sizeof(int));
...
// Speicher auf 50 int-Element verkleinert
ptr = realloc( ptr, 50 * sizeof(int));
```

Vergrößern Sie hingegen den Speicherblock, müssen Sie immer die vorhandene Blockgröße des bereits vorhandenen Speicherblocks in Bytes mitrechnen. Folgendes ist beispielsweise falsch:

```
// Speicher für 100 int-Elemente reserviert
int block = 256;
ptr = malloc( block * sizeof(int) );
// Hier wird kein neuer Speicherplatz reserviert, es wird
// lediglich erneut Speicher für 256 int-Elemente reserviert.
ptr = realloc( ptr, block * sizeof(int) );
```

Damit der Speicherblock wirklich vergrößert wird, muss die Angabe des zweiten Parameters auch tatsächlich der gesamten Speichergröße des Blocks im Heap entsprechen. Korrekt wäre daher:

```
int block = 256;
// Speicher für 256 int-Elemente reservieren
ptr = malloc( block * sizeof(int) );
...
block += block;
// Speicher für insgesamt 512 int-Elemente anfordern.
// Insgesamt wurde der Speicher um 256 int-Elemente vergrößert.
ptr = realloc( ptr, block * sizeof(int) );
```

Hierzu ein einfaches Beispiel, das `realloc()` in der Praxis demonstrieren soll:

```
00  // kap011/listing003.c
01  #include <stdio.h>
02  #include <stdlib.h>
03  #define BLKSIZE 8

04  int main(void) {
05    int n=0, max=BLKSIZE, z=0;
06    int* zahlen = calloc(BLKSIZE, sizeof(int));
07    if(NULL == zahlen) {
08      printf("Kein virtueller RAM mehr vorhanden!\n");
09      return EXIT_FAILURE;
10    }
11    printf("Zahlen eingeben --- Beenden mit 0\n");
12    while(1) {
13      printf("Zahl (%d) eingeben: ", n+1);
```

```
14      if( scanf("%d", &z) != 1 ) {
15        printf("Fehler bei der Eingabe\n");
16        return EXIT_FAILURE;
17      }
18      if(z==0) { break; } // Schleifenabbruch
19      if(n >= max-1) {
20        max += BLKSIZE;
21        zahlen = realloc(zahlen, max * sizeof(int) );
22        if(NULL == zahlen) {
23          printf("Kein virtueller RAM mehr vorhanden!");
24          return EXIT_FAILURE;
25        }
26        printf("Neuer Speicher: %d Bytes\n", BLKSIZE);
27        printf("Insgesamt     : %zd Bytes\n",
             sizeof(int)*max);
28        printf("Platz fuer    : %d Elemente\n", max);
29      }
30      zahlen[n++] = z;
31    }
32    printf("Folgende Zahlen wurden eingegeben ->\n\n");
33    for(int i = 0; i < n; i++) {
34      printf("%d ", zahlen[i]);
35    }
36    printf("\n");
37    free(zahlen);
38    return EXIT_SUCCESS;
39  }
```

Zuerst wird in Zeile **(06)** ein Speicherblock für `BLKSIZE` `int`-Elemente mit `calloc()` angelegt. Sie können hier genauso gut `malloc()` verwenden. Anschließend geben Sie in der `while`-Schleife (Zeilen **(12)** bis **(31)**) `BLKSIZE` `int`-Elemente ein. In Zeile **(06)** haben Sie dafür einen Speicherblock vom Heap reserviert. Mit der Eingabe von 0 können Sie die Schleife abbrechen. Das wird in Zeile **(18)** überprüft. In Zeile **(19)** wird regelmäßig kontrolliert, ob noch Speicherplatz für weitere Elemente vorhanden ist. Ist das nicht der Fall, wird in `if` verzweigt (Zeilen **(20)** bis **(25)**). Zunächst muss in Zeile **(20)** die alte Größe des Speicherblocks mit dem neu zu reservierenden

Speicherplatz addiert werden. Dann wird in Zeile **(21)** der Speicherplatz um `BLKSIZE` Elemente vom Typ `int` erweitert, um weitere Elemente einlesen zu können.

Das Programm bei der Ausführung:

```
Zahlen eingeben --- Beenden mit 0
Zahl (1) eingeben: 123
Zahl (2) eingeben: 234
Zahl (3) eingeben: 345
Zahl (4) eingeben: 456
Zahl (5) eingeben: 678
Zahl (6) eingeben: 789
Zahl (7) eingeben: 912
Zahl (8) eingeben: 345
Neuer Speicher: 8 Bytes
Insgesamt     : 64 Bytes
Platz für     : 16 Elemente
Zahl (9) eingeben: 321
Zahl (10) eingeben: 432
Zahl (11) eingeben: 543
Zahl (12) eingeben: 0
Folgende Zahlen wurden eingegeben ->
123 234 345 456 678 789 912 345 321 432 543
```

11.3 Speicherblock freigeben

Wenn Sie Speicher vom Heap angefordert haben und nicht mehr benötigen, müssen Sie diesen Speicher wieder für das System freigeben. In C muss die Speicherfreigabe explizit mit der Funktion `free()` durchgeführt werden. Hier die Syntax von `free()`:

```
#include <stdlib.h>
void free( void* ptr );
```

Damit geben Sie den dynamisch zugeteilten Speicherblock frei, den Sie mit Funktionen wie `malloc()`, `calloc()` oder `realloc()` angefordert haben und auf dessen Adresse der Zeiger `ptr` verweist. Ist `ptr` ein `NULL`-Zeiger,

passiert gar nichts. Da der Speicherbereich mit `free()` freigegeben wurde, kann er bei späteren Speicheranforderungen wiederverwendet werden.

Bei dem Argument `ptr` müssen Sie selbst darauf achten, dass auch wirklich ein Zeiger verwendet wird, der zuvor mit einer Funktion wie `malloc()`, `calloc()` oder `realloc()` reserviert wurde. Verwenden Sie einen falschen Zeiger oder wurde der Speicher bereits freigegeben, lässt sich das weitere Verhalten des Programms nicht mehr vorhersagen und ist undefiniert.

Speicher wird beim Programmende automatisch freigegeben

Es wird generell behauptet, dass bei der Beendigung eines Programms das Betriebssystem den reservierten und nicht mehr freigegebenen Speicher selbst organisiert und somit auch wieder freigibt. Meistens ist das auch der Fall, vom Standard ist das aber nicht gefordert. Somit hängt dieses Verhalten von der Implementierung der Speicherverwaltung des Betriebssystems ab.

Memory Leaks (Speicherlecks)

Memory Leaks sind Speicherbereiche, die zwar belegt sind, aber zur Laufzeit weder verwendet noch freigegeben werden können. Ein populäres Beispiel ist die Reservierung von Speicher mittels `malloc()`. Auf die Anfangsadresse des Speicherblocks verweist ein Zeiger. Geht dieser Zeiger verloren, weil Sie ihn beispielsweise anderweitig verwenden oder bereits erneut einen Speicher mit `malloc()` reservieren, gibt es keine Möglichkeit mehr, auf diesen Speicherbereich zuzugreifen. Er kann somit auch nicht mehr freigegeben werden. In C gibt es keinen standardisierten Weg der automatischen Speicherbereinigung. Er muss explizit mit `free()`freigegeben werden.

In diesem einfachen Beispiel wird ein solches Speicherleck demonstriert:

```
00  // kap011/listing004.c
01  #include <stdio.h>
02  #include <stdlib.h>

03  int main(void) {
04    double *dptr1=NULL, *dptr2=NULL;
```

```
05    dptr1 = calloc( 10, sizeof(double) );
06    dptr2 = calloc( 20, sizeof(double) );
07    // dptr1 verweist auf dieselbe Adresse wie dptr2
08    dptr1 = dptr2;          // Speicherleck erstellt
09    //... nach vielen Zeilen Code Speicher freigeben
10    free( dptr1 );  // Gibt Speicher frei
11    free( dptr2 );  // Fehler!!!
12    return EXIT_SUCCESS;
13  }
```

In Zeile **(05)** und **(06)** wird jeweils ein Speicherblock für 10 bzw. 20 `double`-Werte reserviert. In Zeile **(08)** verweisen Sie den Zeiger `dptr1` auf dieselbe Adresse wie `dptr2`. Durch die Zuweisung haben Sie keine Möglichkeit mehr, auf den Speicherbereich zuzugreifen, den Sie mit Zeile **(05)** reserviert haben. In einem dauerlaufenden Programm, wie dies beispielsweise bei Serveranwendungen der Fall ist, kann der Speicherblock nicht mehr vom Programm freigegeben werden. In diesem Beispiel ist das Speicherleck zwar trivial, aber es deckt auf, was passieren kann, wenn man unvorsichtig mit Zeigern umgeht. In Zeile **(11)** machen Sie außerdem noch einen weiteren Fehler, indem Sie versuchen, denselben Speicherblock freizugeben, den Sie bereits in Zeile **(10)** freigegeben haben. Das weitere Verhalten des Programms ist dadurch undefiniert.

Folgendes Beispiel enthält ebenfalls ein Speicherleck. In einer Endlosschleife wird mittels `malloc()` ständig ein Speicherblock angefordert, aber niemals freigegeben. Da immer wieder derselbe Zeiger verwendet wird, kann nach jeder Speicheranforderung nicht mehr auf den zuvor reservierten Speicherbereich zurückgegriffen werden. Damit kann er auch nicht freigegeben werden. Der Speicher läuft voll, bis das Programm nicht mehr reagiert:

```
double *dptr1=NULL;
...
while( 1 ) {
  dptr1 = malloc( 10 * (sizeof(double)));
  ...
}
free(dptr1)  // Wird vielleicht nie erreicht
```

Das Problem in diesem Codeausschnitt können Sie beheben, indem Sie gleich am Ende der `while`-Schleife den Speicherblock wieder freigeben:

```
double *dptr1=NULL;
...
while( 1 ) {
  dptr1 = malloc( 10 * (sizeof(double)));
  ...
  free(dptr1);  // Speicherleck stopfen
}
```

11.4 Kontrollfragen und Aufgaben

1. Welche Funktionen stehen Ihnen zur Verfügung, wenn Sie die Laufzeit eines Speichers vom Heap anfordern möchten? Beschreiben Sie kurz die Unterschiede der einzelnen Funktionen.
2. Welchen Typ geben alle Funktionen zur dynamischen Speicherreservierung zurück?
3. Wie können Sie nicht benötigten Speicher wieder an das System zurückgeben?
4. Im folgenden Codeausschnitt wurde ein übler Fehler gemacht. Welcher?

```
01    int *iarray1=NULL, *iarray2=NULL;
02    iarray1 = malloc( BLK * sizeof(int) );
03    iarray2 = malloc( BLK * sizeof(int) );
...
04    for(int i=0; i<BLK; i++) {
05      iarray1[i] = i;
06      iarray2[i] = i+i;
07    }
08    iarray1 = iarray2;
09    iarray2 = iarray1;
...
```

5. Erstellen Sie ein Programm, das einen String (`char`-Array) mit einer unbestimmten Länge einlesen kann. **Tipp:** Lesen Sie den String zunächst in einen statischen Puffer ein, zählen Sie die Zeichen, und reservieren Sie dann Speicher dafür. Hängen Sie ggf. den neuen Text hinten an.
6. Schreiben Sie eine Funktion, die ermittelt, ob ein Funktionsaufruf von `realloc()` den kompletten Speicherblock umkopieren musste. Im Grunde müssen Sie nur die Adressen sichern und miteinander vergleichen. Geben Sie im Falle eines Umkopierens die alte und die neue Speicheradresse (mit `%p`) auf dem Bildschirm aus.

Kapitel 12

Komplexe Datentypen

Wenn Sie auf der Suche nach einer komplexeren Datenstruktur sind, die unterschiedliche, aber logisch zusammenhängende Typen speichern kann, dann sind Sie bei den Strukturen genau richtig. Mit den Strukturen können Sie solche Daten selbst modellieren.

Neben Strukturen können Sie auch eine Union bilden. Eine Union ist wie eine Struktur aufgebaut, nur haben die einzelnen Elemente (engl. *members*) einer Union immer die gleiche Startadresse. Folglich können Sie bei einer Union zwar ebenfalls mehrere Datentypen zusammenfassen, aber es kann immer nur ein Element genutzt werden.

Mit dem Aufzählungstyp `enum` können Sie den Elementen mit einem Aufzählungstyp einen Namen zuordnen und sie mit diesem Namen verwenden. Intern sind die Aufzählungstypen als Ganzzahlen codiert.

Selbst definierte Datentypen

Somit können Sie mit C einen selbst definierten Datentyp aus einer Struktur (`struct`), einer Union (`union`) und dem Aufzählungstyp (`enum`) erstellen.

Einen Aliasnamen für existierende Datentypen oder Namen zu einem vereinbarten aggregierten Datentyp (z. B. einer Struktur) können Sie mit `typedef` vereinbaren.

12.1 Strukturen

Mit den Strukturen können Sie einen eigenen Typ definieren, in dem Sie die einzelnen Daten und die Eigenschaften bestimmter Objekte beschreiben und zusammenfassen (z. B. Kundendaten einer Firma). Sie werden zu Datensätzen (engl. *records*) zusammengefasst. Der Datensatz selbst

besteht wiederum aus Datenfeldern mit den nötigen Informationen (Name, Adresse, Telefonnummer usw.). Datenfelder sind hier die Strukturvariablen und somit die Elemente einer Struktur.

12.1.1 Strukturtyp deklarieren

Die Deklaration eines Strukturtyps fängt immer mit dem Schlüsselwort `struct` an und enthält eine Liste von Elementen in geschweiften Klammern. Die Liste wird mit einem Semikolon abgeschlossen. Hierbei muss mindestens ein Element deklariert werden. Die Syntax einer solchen Struktur sieht folgendermaßen aus:

```
struct [bezeichner1] { Liste_von_Komponenten } [bezeichner2];
```

Die Komponenten der Struktur (auch *Member* oder *Membervariablen* genannt) können einen beliebigen Datentyp enthalten, der natürlich auch zuvor definierte Strukturtypen beinhalten kann. Nicht erlaubt als Komponenten einer Struktur sind Funktionen.

Mit dem folgenden Strukturtyp `Id3_tag` werden Daten ähnlich einer MP3-Datei beschrieben:

```
struct Id3_tag {
  char titel[30];
  char kuenstler[30];
  char album[30];
  short jahr;
  char kommentar[30];
  char genre[30];
};
```

Mit dieser Typvereinbarung haben Sie einen *Strukturtyp* `Id3_tag` erstellt, der sich aus sechs Komponentenvariablen zusammensetzt und die typischen Informationen für ein MP3-Musikstück enthält.

ID3-Tag

ID3-Tags sind die Informationen (Metadaten), die in den MP3-Audiodateien enthalten sind (ID3 = **Id**entify an MP**3**). Um allerdings vor falschen

Erwartungen zu warnen: Die Struktur im Beispiel ist nicht völlig ID3-Tag-konform. Die einzelnen Datentypen (Länge) wurde hier nicht ganz eingehalten. Falls Sie vorhaben, einen echten ID3-Tag-Editor zu erstellen bzw. die Tags auszulesen, finden Sie unter *http://www.id3.org/* und unter *https://de.wikipedia.org/wiki/ID3-Tag* weitere Informationen zu diesem Thema.

Wie bei den Basisdatentypen können Sie bei der Typvereinbarung Komponentenvariablen mit demselben Typ in der Struktur, getrennt durch ein Komma, zusammenfassen. Hier ein Beispiel:

```
struct Id3_tag {
  char titel[30], kuenstler[30], album[30];
  short jahr;
  char kommentar[30], char genre[30];
};
```

Hier soll nochmals erwähnt werden, dass bisher nur die Rede von einer Deklaration ist, nicht aber von der Speicherplatzreservierung. Speicherplatz wird erst reserviert, wenn Sie eine Strukturvariable, also einen Bezeichner, vom Strukturtyp definieren.

12.1.2 Definition einer Strukturvariablen

Die Definition einer Struktur erstellen Sie mithilfe eines Bezeichners. In diesem Fall spricht man von der Definition einer *Strukturvariablen*. Folgendermaßen werden beispielsweise drei Strukturvariablen mit dem Typ `struct Id3_tag` definiert. Dabei wird Speicherplatz für den Typ `struct Id3_tag` und seine einzelnen Komponentenvariablen reserviert:

```
// Definition einer Strukturvariablen
struct Id3_tag data1;
...
// Definition von zwei Strukturvariablen
struct Id3_tag data2, data3
```

12.1.3 Erlaubte Operationen auf Strukturvariablen

Folgende Operationen sind mit Strukturvariablen erlaubt:

- Zuweisen einer Struktur an eine andere Struktur mit demselben Typ
- Rückgabe und Übergabe von Strukturen von einer und an eine Funktion
- Ermitteln der Adresse einer Struktur mit dem Adressoperator `&`
- Ermitteln der Größe einer Struktur mit dem `sizeof`-Operator und die Speicheranordnung mit dem `Alignof`-Operator (seit C11)
- Selektion einzelner Komponenten

Auf die Details der einzelnen Operationen wird auf den folgenden Seiten ausführlicher eingegangen.

12.1.4 Deklaration und Definition zusammenfassen

Die Deklaration eines Strukturtyps und die Definition einer Strukturvariablen können Sie auch zu einer Anweisung zusammenfassen:

```
struct Id3_tag {
  char titel[30];
  char kuenstler[30];
  char album[30];
  short jahr;
  char kommentar[30];
  char genre[30];
} data;
```

Hiermit deklarieren Sie den Strukturtyp `struct Id3_tag` und definieren eine Strukturvariable namens `data`. Sie können auch mehrere Strukturvariablen auf einmal definieren:

```
struct Id3_tag {
  char titel[30];
  char kuenstler[30];
  char album[30];
  short jahr;
```

```
  char kommentar[30];
  char genre[30];
} data1, data2, data3, data4, data5;  // 5 Strukturvariablen
```

12.1.5 Synonyme für Strukturtypen erstellen

Wenn Sie nicht ständig das Schlüsselwort `struct` bei der Definition der Strukturvariablen verwenden möchten, können Sie mit `typedef` ein Synonym dafür erstellen. Hier ein Beispiel:

```
typedef struct Id3_tag Id3_t;
```

Durch `typedef` können Sie jetzt eine Struktur mit einer Strukturvariablen folgendermaßen definieren:

```
Id3_t data1;   // Anstatt: struct Id3_tag data1;
```

Ein solches Synonym können Sie bei der Deklaration des Strukturtyps wie folgt erstellen:

```
typedef struct Id3_tag {
  char titel[30];
  char kuenstler[30];
  char album[30];
  short jahr;
  char kommentar[30];
  char genre[30];
} Id3_t;
```

Statt `struct Id3_tag` verwenden jetzt Sie nur noch `Id3_t` für den Strukturtyp.

12.1.6 Selektion auf Komponenten einer Strukturvariablen

Um auf die Komponentenvariablen einer Strukturvariablen zugreifen zu können, werden der Punkt-Operator und der Pfeil-Operator verwendet. Auf den Pfeil-Operator wird noch gesondert in Abschnitt 12.1.12, Unterabschnitt »Selektion der Komponenten über den Pfeil-Operator«, eingegan-

gen. Hier wird zunächst der Punkt-Operator behandelt. Dazu ein einfaches Beispiel:

```
Id3_t data;
...
strncpy(data.titel, "21st Century Breakdown", 29);
strncpy(data.kuenstler, "Green Day", 29);
...
data.jahr=2009;
...
```

Sie speichern die Strings `"21st Century Breakdown"` und `"Green Day"` in den Komponentenvariablen `titel` und `kuenstler`. Der Komponentenvariablen `jahr` übergeben Sie den Wert 2009.

Nachfolgend ein einfaches Beispiel, wie Sie den einzelnen Komponenten einer Strukturvariablen einen Inhalt hinzufügen und darauf zugreifen können:

```
00  // kap012/listing001.c
01  #include <stdio.h>
02  #include <stdlib.h>
03  #define MAX 64

04  typedef struct Id3_tag {
05    char titel[MAX];
06    char kuenstler[MAX];
07    char album[MAX];
08    short jahr;
09    char kommentar[MAX];
10    char genre[MAX];
11  } Id3_t;

12  void output(Id3_t song) {
13    printf("\n\nDer Song, den Sie eingaben:\n");
14    printf("Titel     :  %s", song.titel);
15    printf("Künstler  :  %s", song.kuenstler);
16    printf("Album     :  %s", song.album);
```

```
17    printf("Jahr      :  %hd\n", song.jahr);
18    printf("Kommentar :  %s", song.kommentar);
19    printf("Genre     :  %s", song.genre);
20  }

21  void dump_buffer(FILE *fp) {
22    int ch;
23    while( (ch = fgetc(fp)) != EOF && ch != '\n' )
24      /* Kein Anweisungsblock nötig */ ;
25  }

26  int main(void) {
27    Id3_t data;
28    printf("Titel     :  ");
29    if(fgets(data.titel,sizeof(data.titel),stdin)==NULL) {
30      printf("Fehler bei der Eingabe\n");
31      return EXIT_FAILURE;
32    }
33    printf("Kuenstler :  ");
35    if(fgets(data.kuenstler,sizeof(data.kuenstler),stdin)
            ==NULL){
36      printf("Fehler bei der Eingabe\n");
37      return EXIT_FAILURE;
38    }
39    printf("Album     :  ");
40    if(fgets(data.album,sizeof(data.album),stdin)==NULL) {
41      printf("Fehler bei der Eingabe\n");
42      return EXIT_FAILURE;
43    }
44    printf("Jahr      :  ");
45    if (scanf("%hd", &data.jahr) != 1 ) {
46      printf("Fehler bei der Eingabe\n");
47      return EXIT_FAILURE;
48    }
49    dump_buffer(stdin);
50    printf("Kommentar :  ");
```

```
51    if(fgets(data.kommentar,sizeof(data.kommentar),stdin)
            == NULL ) {
52      printf("Fehler bei der Eingabe\n");
53      return EXIT_FAILURE;
54    }
55    printf("Genre    :  ");
56    if(fgets(data.genre,sizeof(data.genre),stdin)==NULL) {
57      printf("Fehler bei der Eingabe\n");
58      return EXIT_FAILURE;
59    }
60    // Daten der Struktur ausgeben
61    output(data);
62    return EXIT_SUCCESS;
63  }
```

In Zeile **(27)** wird die Strukturvariable `data` vom Typ `struct Id3_tag` definiert, und in den Zeilen **(28)** bis **(59)** werden die einzelnen Elemente der Struktur mithilfe des Punkt-Operators eingelesen. Die Funktion `dump_buffer()` in der Zeile **(49)** ist ein Workaround, um ein verbleibendes Newline-Zeichen aus dem Eingabepuffer nach `scanf()` aus der Standardeingabe (`stdin`) zu holen, damit dieses in der Standardeingabe befindliche Newline-Zeichen des Tastaturpuffers nicht für das nächste `fgets()` in der Zeile **(51)** verwendet wird und ohne eine weitere Eingabe zum übernächsten `fgets()` in der Zeile **(56)** springen würde. Die Funktion wird in den Zeilen **(21)** bis **(25)** definiert.

Achtung vor »fflush(stdin)«

Anstelle des selbstgeschriebenen Workarounds der Funktion `dump_buffer()` wird häufig die falsche Lösung mit `fflush(stdin)` verwendet. Auch wenn diese Funktion bei einigen Compilern prima funktioniert, ist diese Möglichkeit zum einen nicht portabel, und zum anderen ist das Verhalten nach dem C-Standard undefiniert, weil die Funktion `fflush()` für Ausgabestreams und nicht für Eingabestreams gedacht ist.

In Zeile **(61)** wird die komplette Struktur als Kopie an die Funktion `output()` (Zeilen **(12)** bis **(20)**) übergeben. Dort wird demonstriert, wie Sie mithilfe

des Punkt-Operators die einzelnen Komponenten der Strukturvariablen ausgeben können.

Das Programm bei der Ausführung:

```
Titel     :  Heavy Cross
Künstler  :  Gossip
Album     :  Heavy Cross - Single
Jahr      :  2009
Kommentar :  Cooler Punk-Rock
Genre     :  Alternative

Der Song, den Sie eingaben:
Titel     :  Heavy Cross
Künstler  :  Gossip
Album     :  Heavy Cross - Single
Jahr      :  2009
Kommentar :  Cooler Punk-Rock
Genre     :  Alternative
```

12.1.7 Strukturen initialisieren

Sie initialisieren eine Strukturvariable bei der Definition explizit, indem Sie eine Initialisierungsliste verwenden, in der die einzelnen Initialisierer durch ein Komma getrennt in geschweiften Klammern zugewiesen werden. Halten Sie hier die richtige Reihenfolge der Deklaration der Struktur ein. Ebenso muss der Initialisierer demselben Typ der Deklaration entsprechen. Ein kurzer Codeausschnitt zeigt Ihnen, wie dies gemacht werden kann:

```
Id3_t data = {
    "Kings and Queens",
    "30 Seconds to Mars",
    "This Is War",
    2009,
    "Komponist: Jared Leto",
    "Alternative"
};
```

In der Liste enthält jede Komponentenvariable einen Initialisierer. Sie können aber auch weniger Initialisierer angeben, als es Komponentenvariablen in der Struktur gibt. Dennoch müssen Sie die richtige Reihenfolge und den Typ einhalten. Hier ein Beispiel:

```
Id3_t data = {
    "Kings and Queens",
    "30 Seconds to Mars",
};
```

Hier werden nur die ersten beiden Komponentenvariablen der Strukturvariablen initialisiert. Die restlichen Komponenten werden automatisch mit 0 bzw. `NULL` initialisiert. Sie können quasi mit einer leeren Initialisierungsliste alle Werte der Komponentenvariablen mit 0 vorbelegen.

12.1.8 Nur bestimmte Komponenten einer Strukturvariablen initialisieren

Seit dem C99-Standard ist es möglich, nur bestimmte Komponenten einer Strukturvariablen zu initialisieren. Als Initialisierer muss ein sogenannter *Elementebezeichner* verwendet werden, der wie folgt aussieht:

```
.strukturelement = wert
```

Ein Beispiel hierzu, bezogen auf die Struktur `Id3_t` (alias `struct Id3_tag`):

```
Id3_t data = {
   .kuenstler = "Stanfour",
   .album  = "Wishing Yo Well",
   .genre = "Rock"
};
```

Hier wurden die Elemente `kuenstler`, `album` und `genre` der Strukturvariablen initialisiert. Die restlichen Elemente werden automatisch mit 0 initialisiert. Der Clou dabei ist, dass Sie nicht einmal auf die richtige Reihenfolge der Elemente achten müssen. Folgende Initialisierung verursacht keinerlei Probleme:

```
Id3_t data = {
    .genre = "Pop",
    .jahr = 2010,
    .kuenstler = "Goldfrapp",
    .titel = "Rocket"
};
```

Sie können bei der Initialisierung aber auch den üblichen Weg gehen und nur für einzelne Komponenten den Elementebezeichner verwenden. Dann müssen Sie allerdings teilweise die Reihenfolge beachten. Hier ein Beispiel:

```
Id3_t data = {
   "Gypsy",
   "Shakira",
   .genre = "Pop"
};
```

Hier werden die ersten zwei Elemente wie mit einer üblichen Initialisierungsliste initialisiert und das letzte Element `genre` mithilfe des Elementebezeichners. Die restlichen Werte werden automatisch mit 0 initialisiert.

12.1.9 Zuweisung bei Strukturvariablen

Auch eine Zuweisung von zwei Strukturvariablen vom selben Strukturtyp ist erlaubt. Eine solche Zuweisung kann beispielsweise wie folgt aussehen:

```
Id3_t data = {
    "Kings and Queens",
    "30 Seconds to Mars",
    "This Is War",
    2009,
    "Komponist: Jared Leto",
    "Alternative"
};
Id3_t copyData;
...
copyData = data;
```

Bei einer solchen Zuweisung werden alle Komponentenwerte von `data` an die jeweiligen Komponentenvariablen von `copyData` zugewiesen.

12.1.10 Größe und Speicherausrichtung einer Struktur

Die Größe der Strukturvariablen eines Strukturtyps lässt sich nicht anhand der einzelnen darin enthaltenen Komponentenvariablen errechnen, weil bei dem eben erwähnten Padding-Verfahren (oder auch *Alignment*) die Komponenten auf eine bestimmte Größe »aufgefüllt« werden. Daher sollten Sie auch hier zur Ermittlung der Größe den `sizeof`-Operator verwenden (beispielsweise `sizeof(struct Id3_tag)`). Seit C11 können Sie auch mit dem `_Alignof`-Operator das Alignment der ganzen Struktur ermitteln.

12.1.11 Strukturen vergleichen

Da Strukturen für die Speicherausrichtung (engl. *data alignment*) das sogenannte *Data-Structure-Padding*-Verfahren verwenden (es gibt Lücken zwischen den Komponenten), gibt es keinen portablen Weg, mit dem Sie zwei Strukturvariablen desselben Typs direkt mit dem `==`-Operator vergleichen können. Es bleibt Ihnen also nichts anderes übrig, als selbst eine Funktion zu schreiben, die zwei Strukturen Element für Element miteinander vergleicht.

12.1.12 Strukturen, Funktionen und Strukturzeiger

In unserem ersten Beispiel dieses Kapitels, *listing001.c*, wurde eine Struktur als Kopie an eine Funktion übergeben. Der Vorgang ist allerdings nicht sehr effektiv, weil Strukturen einen ziemlichen Datenumfang haben können. So hat die Struktur `struct Id3_tag` auf einem Testrechner hier beispielsweise 322 Bytes, die bei jedem Funktionsaufruf auf dem Stack kopiert und wieder freigegeben werden müssen. Bei häufigen Funktionsaufrufen wird deshalb die Laufzeit des Programms erheblich beeinträchtigt. Daher ist es oft wesentlich effizienter, hierfür Zeiger zu verwenden.

Zeiger auf Strukturvariablen

Zeiger auf Strukturvariablen eines bestimmten Strukturtyps lassen sich genauso realisieren wie Zeiger auf normalen Variablen von Basisdaten-

typen. Mit folgender Anweisung speichern Sie z. B. die Adresse der Strukturvariablen `data` im Strukturvariablenzeiger `id3_ptr`:

```
struct Id3_tag data = {
    "Kings and Queens",
    "30 Seconds to Mars",
    "This Is War",
    2009,
    "Komponist: Jared Leto",
    "Alternative"
};
struct Id3_tag *id3_ptr = NULL;
// id3_ptr verweist auf die Anfangsadresse von data
id3_ptr = &data;
```

Selektion der Komponenten über den Pfeil-Operator

Wenn Sie einen Strukturvariablen-Zeiger verwenden, welcher die Adresse einer Strukturvariablen vom selben Strukturtyp enthält, erfolgt der Zugriff auf die einzelnen Komponentenvariablen etwas anders. Zwar können Sie auch den Dereferenzierungsoperator und den Punkt-Operator auf die gewünschte Komponente in der Struktur anwenden, aber Sie müssen dann eine Klammerung verwenden, weil der Punkt-Operator eine höhere Priorität als der Dereferenzierungsoperator (*) hat. Ein Zugriff auf eine einzelne Komponente über den Strukturvariablenzeiger müsste daher mit folgender Syntax realisiert werden:

```
(*struct_zeiger).komponente
```

Allerdings ist diese Art, einen Zeiger auf die einzelnen Komponenten zu verwenden, etwas umständlich. Glücklicherweise gibt es mit dem Pfeil-Operator (->) eine einfache und äquivalente Alternative. Die Syntax mit dem Pfeil-Operator sieht wie folgt aus:

```
struct_zeiger->komponente // == (*struct_zeiger).komponente
```

Mit diesen Kenntnissen soll jetzt ein Beispiel erstellt werden, mit dem eine Strukturvariable dynamisch zur Laufzeit in einer Funktion erzeugt und die Adresse an den Aufrufer zurückgegeben wird. Dieser Strukturvari-

ablenzeiger wird außerdem noch an eine Funktion übergeben, und der Inhalt der einzelnen Komponenten wird ausgegeben.

```
00 // kap012/listing002.c
01 #include <stdio.h>
02 #include <stdlib.h>
03 #define MAX 64

04 typedef struct Id3_tag {
     char titel[MAX];
     char kuenstler[MAX];
     char album[MAX];
     short jahr;
     char kommentar[MAX];
     char genre[MAX];
   } Id3_t;

05 void dump_buffer(FILE *fp) {
06   int ch;
07   while( (ch = fgetc(fp)) != EOF && ch != '\n' )
08     /* Kein Anweisungsblock nötig */ ;
09 }

10 void print_id3(Id3_t* song) {
11   if(song == NULL ) {
12     printf("Kein Inhalt gefunden\n");
13   }
14   else {
15     printf("\n\nDer Song, den Sie eingaben:\n");
16     printf("Titel     :  %s", song->titel);
17     printf("Kuenstler :  %s", song->kuenstler);
18     printf("Album     :  %s", song->album);
19     printf("Jahr      :  %hd\n", song->jahr);
20     printf("Kommentar :  %s", song->kommentar);
21     printf("Genre     :  %s", song->genre);
22   }
23 }
```

```
24 Id3_t *new_id3(void) {
25   Id3_t* data = malloc(sizeof(Id3_t));
26   if( data == NULL ) {
27     printf("Fehler bei der Speicheranforderung\n");
28     return NULL;
29   }
30   printf("Titel     :  ");
31   if(fgets(data->titel,sizeof(data->titel),stdin)==NULL) {
32     printf("Fehler bei der Eingabe\n");
33     free(data);  // Speicher vorher freigeben
34     return NULL; // NULL-Zeiger zurück
35   }
36   printf("Kuenstler :  ");
37   if(fgets( data->kuenstler,
              sizeof(data->kuenstler), stdin) == NULL ) {
38     printf("Fehler bei der Eingabe\n");
39     free(data);
40     return NULL;
41   }
42   printf("Album     :  ");
43   if(fgets(data->album,sizeof(data->album),stdin)==NULL) {
44     printf("Fehler bei der Eingabe\n");
45     free(data);
46     return NULL;
47   }
48   printf("Jahr      :  ");
49   if (scanf("%hd", &data->jahr) != 1 ) {
50     printf("Fehler bei der Eingabe\n");
51     free(data);
52     return NULL;
53   }
54   dump_buffer(stdin);
55   printf("Kommentar :  ");
56   if(fgets( data->kommentar,
              sizeof(data->kommentar), stdin) == NULL ) {
```

```
57      printf("Fehler bei der Eingabe\n");
58      free(data);
59      return NULL;
60    }
61    printf("Genre    :  ");
62    if(fgets(data->genre,sizeof(data->genre),stdin)==NULL){
63      printf("Fehler bei der Eingabe\n");
64      free(data);
65      return NULL;
66    }
67    return data;
68  }

69  int main(void) {
70    Id3_t *data = new_id3();
71    if( data != NULL ) {
72      print_id3(data);
73    }
74    free(data); // auch im Fall von NULL kein Problem
75    return EXIT_SUCCESS;
76  }
```

In Zeile **(70)** rufen Sie die Funktion `new_id3()` auf, deren Rückgabewert Sie dem Strukturvariablenzeiger `data` zuweisen. In der Funktion `new_id3()` selbst (Zeilen **(24)** bis **(68)**) wird zunächst in Zeile **(25)** ein Speicher vom Heap für eine neue Struktur angefordert. Anschließend werden in den Zeilen **(30)** bis **(66)** die einzelnen Elemente der dynamisch reservierten Strukturvariablen eingelesen. Das haben Sie bereits recht ähnlich im ersten Beispiel (*listing001.c*) dieses Kapitels gesehen. Hier wird jetzt allerdings der Pfeil-Operator statt des Punkt-Operators für den selektiven Zugriff auf die einzelnen Komponenten der Strukturvariablen verwendet. In Zeile **(67)** wird die Anfangsadresse der kompletten Struktur an den Aufrufer (Zeile **(70)**) zurückgegeben.

In Zeile **(72)** wird die Adresse an die Funktion `print_id3()` (Zeilen **(10)** bis **(23)**) übergeben. In der Funktion `print_id3()` selbst wird dann nochmals demonstriert, wie mithilfe des Pfeil-Operators auf die einzelnen Kompo-

nenten zugegriffen werden kann. Das Programm bei der Ausführung entspricht ansonsten dem Beispiel *listing001.c*. zuvor.

12.1.13 Array von Strukturvariablen

Im Grunde spricht nichts dagegen, ein Array von Strukturvariablen eines Strukturtyps zu verwenden. Auch diese lassen sich nutzen wie Arrays mit gewöhnlichen Basisdatentypen. Für den Zugriff auf die einzelnen Komponentenvariablen wird hier der Punkt-Operator verwendet. Ein einfaches Beispiel dazu:

```
00 // kap012/listing003.c
01 #include <stdio.h>
02 #include <stdlib.h>
03 #define MAX 64
04 #define SIZE 3

05 typedef struct Id3_tag {
     char titel[MAX];
     char kuenstler[MAX];
     char album[MAX];
     short jahr;
     char kommentar[MAX];
     char genre[MAX];
   } Id3_t;

06 void dump_buffer(FILE *fp) {
07   int ch;
08   while( (ch = fgetc(fp)) != EOF && ch != '\n' )
09     /* Kein Anweisungsblock nötig */ ;
10 }

11 void print_id3(Id3_t* song) {
12   if(song == NULL ) {
13     printf("Kein Inhalt gefunden\n");
14   }
15   else {
16     printf("\n\nAusgabe der ID3-Daten:\n");
```

```
17      printf("Titel     :  %s", song->titel);
18      printf("Kuenstler :  %s", song->kuenstler);
19      printf("Album     :  %s", song->album);
20      printf("Jahr      :  %hd\n", song->jahr);
21      printf("Kommentar :  %s", song->kommentar);
22      printf("Genre     :  %s", song->genre);
23    }
24  }

25  int main(void) {
26    Id3_t data[SIZE];
27    for(int i=0; i<SIZE; i++) {
28      printf("Titel     :  ");
29      if(fgets( data[i].titel,
                  sizeof(data[i].titel),stdin) == NULL ) {
30        printf("Fehler bei der Eingabe\n");
31        return EXIT_FAILURE;
32      }
33      printf("Kuenstler :  ");
34      if(fgets(data[i].kuenstler,
                 sizeof(data[i].kuenstler), stdin) == NULL ) {
35        printf("Fehler bei der Eingabe\n");
36        return EXIT_FAILURE;
37      }
38      printf("Album     :  ");
39      if( fgets(data[i].album,
                  sizeof(data[i].album), stdin) == NULL ) {
40        printf("Fehler bei der Eingabe\n");
41        return EXIT_FAILURE;
42      }
43      printf("Jahr      :  ");
44      if (scanf("%hd", &data[i].jahr) != 1 ) {
45        printf("Fehler bei der Eingabe\n");
46       return EXIT_FAILURE;
47      }
48      dump_buffer(stdin);
49      printf("Kommentar :  ");
```

```
50      if( fgets(data[i].kommentar,
                  sizeof(data[i].kommentar), stdin) == NULL ) {
51        printf("Fehler bei der Eingabe\n");
52        return EXIT_FAILURE;
53      }
54      printf("Genre    :  ");
55      if( fgets(data[i].genre,
                  sizeof(data[i].genre), stdin) == NULL ) {
56        printf("Fehler bei der Eingabe\n");
57        return EXIT_FAILURE;
58      }
59    }
60    for(int i = 0; i<SIZE; i++ ) {
61      print_id3(&data[i]);
62    }
63    return EXIT_SUCCESS;
64  }
```

In Zeile **(26)** wurde ein Array mit `SIZE`-Elementen vom Strukturtyp `struct Id3_tag` definiert. Die Werte der einzelnen Strukturelemente werden in den Zeilen **(27)** bis **(59)** übergeben. In den Zeilen **(60)** bis **(62)** werden die eingegebenen Werte von `SIZE`-Strukturvariablen ausgegeben. Hierbei wird die im Listing zuvor erstellte Funktion `print_id3()` (Zeilen **(11)** bis **(24)**) verwendet, in der die Anfangsadresse des entsprechenden Elements im Array an die Funktion übergeben wird.

Die Verwendung von Arrays stellt keine allzu großen Anforderungen an den Programmierer; sie sind einfach zu erstellen und zu handhaben. Allerdings sollten Sie sich immer vor Augen halten, dass Sie hiermit eine Menge Speicherplatz vergeuden können. Verwenden Sie ein Array für 1.000 Songtitel (was nicht unbedingt sehr viel ist), wird Speicher für 1.000 Titel verwendet, auch wenn vielleicht im Augenblick nur 100 benötigt werden. Und reichen dann 1.000 Songtitel nicht aus, müssen Sie sich wieder Gedanken über neuen Speicherplatz machen. Bei solchen Programmen sollten Sie den Speicher zur Laufzeit reservieren. Mehr dazu erfahren Sie in Kapitel 11, »Dynamische Speicherverwaltung«.

12.1.14 Strukturvariablen als Komponente in Strukturen

Natürlich spricht überhaupt nichts dagegen, Strukturvariablen als Komponente in einem anderen Strukturtyp zu verwenden. Ein einfaches Beispiel:

```
typedef struct Details {
  short bitrate;
  short spielzeit;
  short abtastrate;
  char version[MAX];
} Info_t;

typedef struct Id3_tag {
  char titel[MAX];
  char kuenstler[MAX];
  char album[MAX];
  short jahr;
  char kommentar[MAX];
  char genre[MAX];
  Info_t info;
} Id3_t;
```

Der Strukturtyp `struct Details` (alias `Info_t`) wird als Komponentenvariable innerhalb der Struktur `struct Id3_tag` (alias `Id3_t`) verwendet. Um beispielsweise über eine Strukturvariable vom Typ `Id3_t` auf die Komponente `bitrate` vom Strukturtyp `Info_t` zugreifen zu können, müssen Sie im Fall einer Strukturvariable einen Punkt-Operator mehr verwenden. Dazu ein Beispiel:

```
Id3_t data;
...
data.info.bitrate = 256;
```

Ist `data` hingegen ein Zeiger, für den zuvor Speicher reserviert wurde, müssen Sie wie gehabt den Pfeil-Operator an der richtigen Stelle verwenden:

```
data->info.bitrate = 256;
```

Es spricht auch nichts dagegen, die beiden Strukturtypen `Id3_t` und `Info_t` als Komponenten in einen neuen Strukturtyp zu packen:

```
typedef struct Meta_Info {
  Id3_t id3;
  Info_t info;
} Meta_t;
```

Abgesehen davon, dass Sie hierbei einen Operator zur Selektion der Komponenten mehr benötigen, ändert sich nicht sehr viel. Sehen Sie sich nachfolgend ein etwas umfangreicheres, aber trotzdem noch einfach gehaltenes Beispiel an, wie Sie auf die einzelnen Komponenten von mehrfach zusammengesetzten Strukturen zugreifen können:

```
00  // kap012/listing004.c
01  #include <stdio.h>
02  #include <stdlib.h>
03  #define MAX 64

04  typedef struct Details {
      unsigned short bitrate;
      unsigned short spielzeit;
      unsigned short abtastrate;
      char version[MAX];
    } Info_t;

05  typedef struct Id3_tag {
      char titel[MAX];
      char kuenstler[MAX];
      char album[MAX];
      short jahr;
      char kommentar[MAX];
      char genre[MAX];
      Info_t info;
    } Id3_t;

06  typedef struct Meta_Info {
```

```
    Id3_t id3;
    Info_t info;
  } Meta_t;

07 void dump_buffer(FILE *fp) {
08   int ch;
09   while( (ch = fgetc(fp)) != EOF && ch != '\n' )
10     /* Kein Anweisungsblock nötig */ ;
11 }

12 void print_id3(Meta_t* song) {
13   if(song == NULL ) {
14     printf("Kein Inhalt gefunden\n");
15   }
16   else {
17     printf("\n\n");
18     printf("Titel     :  %s", song->id3.titel);
19     printf("Kuenstler :  %s", song->id3.kuenstler);
20     printf("Album     :  %s", song->id3.album);
21     printf("Jahr      :  %hd\n", song->id3.jahr);
22     printf("Kommentar :  %s", song->id3.kommentar);
23     printf("Genre     :  %s", song->id3.genre);
24     printf("Bitrate   :  %hu\n", song->info.bitrate);
25     printf("Spielzeit :  %hu\n", song->info.spielzeit);
26     printf("Abtastrate:  %hu\n", song->info.abtastrate);
27     printf("Version   :  %s", song->info.version);
28   }
29 }

30 int main(void) {
31   Meta_t data;
32   printf("Titel     :  ");
33   if( fgets(data.id3.titel,
              sizeof(data.id3.titel), stdin) == NULL ) {
34     printf("Fehler bei der Eingabe\n");
35     return EXIT_FAILURE;
```

```
36    }
37    printf("Kuenstler :  ");
38    if( fgets(data.id3.kuenstler,
                sizeof(data.id3.kuenstler), stdin) == NULL ) {
39      printf("Fehler bei der Eingabe\n");
40      return EXIT_FAILURE;
41    }
42    printf("Album     :  ");
43    if( fgets(data.id3.album,
                sizeof(data.id3.album), stdin) == NULL ) {
44      printf("Fehler bei der Eingabe\n");
45      return EXIT_FAILURE;
46    }
47    printf("Jahr      :  ");
48    if (scanf("%hd", &data.id3.jahr) != 1 ) {
49      printf("Fehler bei der Eingabe\n");
50      return EXIT_FAILURE;
51    }
52    dump_buffer(stdin);
53    printf("Kommentar :  ");
54    if( fgets(data.id3.kommentar,
                sizeof(data.id3.kommentar), stdin) == NULL ) {
55      printf("Fehler bei der Eingabe\n");
56      return EXIT_FAILURE;
57    }
58    printf("Genre     :  ");
59    if( fgets(data.id3.genre,
                sizeof(data.id3.genre), stdin) == NULL ) {
60      printf("Fehler bei der Eingabe\n");
61      return EXIT_FAILURE;
62    }
63    printf("Bitrate   :  ");
64    if (scanf("%hu", &data.info.bitrate) != 1 ) {
65      printf("Fehler bei der Eingabe\n");
66      return EXIT_FAILURE;
67    }
```

```
68    dump_buffer(stdin);
69    printf("Dauer(sec):  ");
70    if (scanf("%hu", &data.info.spielzeit) != 1 ) {
71      printf("Fehler bei der Eingabe\n");
72      return EXIT_FAILURE;
73    }
74    dump_buffer(stdin);
75    printf("Abtastrate:  ");
76    if (scanf("%hu", &data.info.abtastrate) != 1 ) {
77      printf("Fehler bei der Eingabe\n");
78      return EXIT_FAILURE;
79    }
80    dump_buffer(stdin);
81    printf("Version   :  ");
82    if( fgets(data.info.version,
                sizeof(data.info.version), stdin) == NULL ) {
83      printf("Fehler bei der Eingabe\n");
84      return EXIT_FAILURE;
85    }
86    print_id3(&data);
87    return EXIT_SUCCESS;
88  }
```

In der Zeile **(06)** wurde der mehrfach zusammengesetzte Strukturtyp `struct Meta_Info` (alias `Meta_t`) mit den Strukturvariablen vom Typ `Id3_t` und `Info_t` als Komponenten deklariert. In Zeile **(31)** wird die Strukturvariable `data` mit dem Typ `Meta_t` definiert, und in den Zeilen **(32)** bis **(85)** werden die einzelnen Werte an diese verschachtelte Strukturvariable übergeben. Um auch den Zugriff über Zeiger auf die einzelnen Komponenten zu demonstrieren, wurde die Anfangsadresse dieser Struktur in Zeile **(86)** an die Funktion `print_id3()` (Zeilen **(12)** bis **(29)**) übergeben.

Das Programm bei der Ausführung:

```
Titel     :  Use Somebody
Künstler  :  Kings Of Leon
Album     :  Only By The Night
```

```
Jahr      :  2008
Kommentar :  Komponist: Nathan Followill
Genre     :  Alternative Rock
Bitrate   :  256
Dauer(sec):  231
Abtastrate:  44100
Version   :  MPEG-1, Layer 3


Titel     :  Use Somebody
Künstler  :  Kings Of Leon
Album     :  Only By The Night
Jahr      :  2008
Kommentar :  Komponist: Nathan Followill
Genre     :  Alternative Rock
Bitrate   :  256
Spielzeit :  231
Abtastrate:  44100
Version   :  MPEG-1, Layer 3
```

12.1.15 Zeiger als Komponente

An dieser Stelle muss noch kurz auf Zeiger als Komponenten in einem Strukturtyp eingegangen werden – ein etwas komplexeres Thema. Hierbei gilt, dass Zeiger als Komponenten in einer Struktur ebenfalls lediglich wieder eine Anfangsadresse repräsentieren. Verwenden Sie einen Zeiger als Komponente in einer Struktur wie im folgenden Codeausschnitt mit `info`:

```
typedef struct Id3_tag {
  char titel[MAX];
  char kuenstler[MAX];
  char album[MAX];
  short jahr;
  char kommentar[MAX];
  char genre[MAX];
  Info_t *info;  // Zeiger als Komponente
} Id3_t;
```

dann müssen Sie selbst dafür sorgen, dass Sie für die Struktur `Info_t` (alias `struct Details`) einen gültigen Inhalt zuweisen, indem Sie beispielsweise dynamisch Speicher zur Laufzeit reservieren oder eben eine mit Werten belegte Strukturvariable vom Typ `Info_t` mit dem Adressoperator zuweisen. Bezogen auf das Listing *listing004.c* könnten Sie folgendermaßen vorgehen:

```
// kap012/listing005.c
...
Id3_t data;
...
  data.info = malloc(sizeof(Info_t));
  if(data.info != NULL ) {
    printf("Bitrate   :  ");
    if (scanf("%hu", &data.info->bitrate) != 1 ) {
      printf("Fehler bei der Eingabe\n");
      return EXIT_FAILURE;
    }
    dump_buffer(stdin);
    ...
  }
  else {
    data.info = NULL; // Falls kein Speicher reserviert wurde
  }
...
```

Entsprechend erfolgt der Zugriff auf die einzelnen Komponenten mit dem Zeiger in der Struktur über den Pfeil-Operator, wie hier mit `&data.info ->bitrate` demonstriert wurde. Der Zugriff von Strukturvariablenzeigern mit Komponenten als Zeigern in Strukturen erfolgt dann über zwei Pfeil-Operatoren. Bezogen auf das Listing *listing004.c* in der Funktion `print_id3()` sieht dies so aus:

```
// kap012/listing005.c
...
void print_id3(Id3_t* song) {
   if(song == NULL ) {
```

```
    printf("Kein Inhalt gefunden\n");
  }
  else {
    printf("\n\n");
    printf("Titel     :  %s", song->titel);
    printf("Kuenstler :  %s", song->kuenstler);
    printf("Album     :  %s", song->album);
    printf("Jahr      :  %hd\n", song->jahr);
    printf("Kommentar :  %s", song->kommentar);
    printf("Genre     :  %s", song->genre);
    if(song->info != NULL ) {
      printf("Bitrate   :  %hu\n", song->info->bitrate);
      printf("Spielzeit :  %hu\n", song->info->spielzeit);
      printf("Abtastrate:  %hu\n", song->info->abtastrate);
      printf("Version   :  %s", song->info->version);
    }
  }
}
```

Der Vorteil solcher Zeiger in Strukturtypen, die ggf. selbst Strukturen sind, ist, dass sich damit Speicherplatz sparen lässt. Wird beispielsweise diese mehrfach zusammengesetzte Struktur nicht benötigt, so wird auch weniger Speicherplatz benötigt. Nochmals bezogen auf das Listing *listing004.c* könnten Sie somit folgende mehrfach zusammengesetzte Struktur mit Zeigern als Komponenten deklarieren:

```
Typedef struct Meta_Info {
  Id3_t *id3;
  Info_t *info;
} Meta_t;

...
Meta_t data = { NULL, NULL };
```

Natürlich erhöht sich mit Zeigern als Komponenten von zusammengesetzten Strukturen auch die Komplexität des Programms, weil Sie sich zusätzlich noch u. a. eventuell um die Reservierung und Freigabe von

Speicher kümmern müssen. Auch müssen Sie immer sicherstellen, dass Zeiger auf gültige Speicherbereiche verwendet werden.

Achtung bei Zuweisungen

Wenn ein Strukturtyp einen Zeiger als Komponente enthält, müssen Sie bei einer Zuweisung vorsichtig sein, weil hierbei nicht der Inhalt, sondern nur die Adresse des Zeigers kopiert wird, auf die dieser zeigt. Das bedeutet, dass zwei Strukturvariablen mit einem Zeiger als Komponente nach einer Zuweisung auf dieselbe Adresse verweisen.

12.2 Unionen

Eine weitere Möglichkeit, Daten zu strukturieren, sind Unionen (auch Varianten genannt). Ähnlich wie eine Struktur ist auch eine Union ein Verbund verschiedener Datentypen.

Abgesehen von den Schlüsselwörtern `union` und `struct` besteht auf den ersten Blick kein Unterschied zwischen Unionen und Strukturen. Auch die Operationen und der Zugriff auf die Komponentenvariablen erfolgen wie bei den Strukturen.

Beim Speicherplatz allerdings gehen beide dann doch auseinander. Bei Unionen werden die einzelnen Komponenten nicht hintereinander im Speicher abgelegt, sondern alle Komponenten beginnen mit derselben Adresse. Somit werden die einzelnen Komponenten bei einer Union übereinander im Speicher angeordnet.

Dies bedeutet, dass Sie nur eine Komponente in einer Union verwenden können. Daher müssen Sie immer darauf achten, dass Sie die Komponente verwenden, in der Sie zuletzt etwas gespeichert haben. Ein einfaches Beispiel:

```
00  // kap012/listing006.c
01  #include <stdio.h>
02  #include <stdlib.h>
```

```
03 typedef struct rgb {
04   unsigned short red, green, blue;
05 } RGB;

06 typedef union color {
07   RGB rgb;
08   unsigned int hexRGB;
09 } Color;

10 int main(void) {
11   Color color01, color02;
12   color01.rgb.red   = 127;
13   color01.rgb.green = 255;
14   color01.rgb.blue  = 255;
15   color02.hexRGB = 0x7fffff;

16   printf("color01: %x%x%x\n",
       color01.rgb.red,color01.rgb.green,color01.rgb.blue);
17   printf("color02: %x\n", color02.hexRGB);
18   return EXIT_SUCCESS;
19 }
```

In diesem Beispiel wird in den Zeilen (**03**) bis (**05**) zunächst ein Strukturtyp erstellt, in dem Farbwerte für Rot, Grün und Blau gespeichert werden können. In den Zeilen (**06**) bis (**09**) erstellen Sie eine Union mit einer Strukturvariablen vom eben erstellen Strukturtyp `RGB` und einer `unsigned int`-Variablen für den Farbwert als Hexadezimalzahl. In der Praxis können Sie jetzt nur einen der beiden Speicherbereiche der Union verwenden, also entweder `rgb` oder `hexRGB`.

Wenn Sie mit diesem Beispiel etwas experimentieren, werden Sie feststellen, dass es ohne Probleme möglich ist, Code zu kompilieren, der auf die anderen Komponenten einer Union zugreift bzw. den Zugriff versucht. Es findet hierbei keinerlei Überprüfung der Komponenten zur Übersetzungs- und Laufzeit statt. Wie bereits eingangs erwähnt, sind daher Sie als Programmierer dafür verantwortlich, dass auf die richtige Komponente aus der Union zugegriffen wird. Unionen können deshalb eine fehler-

trächtige Sache sein (Stichwort *type punning*). In der Praxis werden die Unionen außer in der Systemprogrammierung oder wenn man komplexere Daten interpretieren will, eher selten verwendet.

Ein einfaches Praxisbeispiel könnte sein, dass ein direkter Zugriff auf die Hardware oder den Speicher notwendig ist. Das könnte wie folgt aussehen:

```
#include <stdint.h> // uint8_t, uint32_t
...
typedef union {
  struct {
     uint8_t byte_1;
     uint8_t byte_2;
     uint8_t byte_3;
     uint8_t byte_4;
  } bytes;
  uint32_t dword;
} HardReg;
...
HardReg reg1, reg2;
reg1.dword = 0x2342341;
...
reg2.byte_2 = 2;
```

Ein weiteres gutes Beispiel könnte eine Schnittstelle zum System sein, wenn Sie eine Verbindung über das Netzwerk, über den USB-Anschluss oder den internen Speicher verwenden wollen. Ein theoretischer Codeausschnitt hierzu:

```
struct DataTransfer {
  unsigned int securityKey;
  unsigned char data[MAXSIZE];
  union {
    struct Network network;
    struct USB usb;
    struct Memory memory;
  }
};
```

Größe einer Union

Es wird häufig irrtümlich angenommen, dass die Größe einer `union` der Größe der größten Komponente in einer `union` entspricht. Auch wenn das in der Praxis bei einem Beispiel einmal zutreffen sollte, ist diese Aussage im Allgemeinen falsch. Der Standard garantiert hier nur, dass die Größe einer `union` gleich `sizeof(union type)` ist.

12.3 Der Aufzählungstyp enum

Mit der `enum`-Aufzählung können Sie Namen für ganzzahlige Datentypen definieren, die Sie dann mit diesen Namen verwenden. Genau genommen sind Aufzählungstypen nichts anderes als Konstanten, weshalb diese häufig auch Aufzählungskonstanten genannt werden. Hier ein einfaches Beispiel:

```
enum religion { rk, evang, islam, bud, sonstige };
```

Hier wird ein neuer Datentyp `religion` mit den Aufzählungskonstanten `rk` (für römisch-katholisch) `evang` (für evangelisch), `islam`, `bud` (für buddhistisch) und `sonstige` definiert. Alle Aufzählungskonstanten sind vom Datentyp `int` und können überall dort eingesetzt werden, wo auch ein `int` verwendet wird. Solche Aufzählungskonstanten werden gerne bei den `case`-Konstanten innerhalb einer `switch`-Anweisung verwendet.

Intern beginnen diese Werte mit der ersten Aufzählungskonstante bei 0, wenn nicht anders vorgegeben, und werden jeweils um den Wert 1 gegenüber dem Vorgänger erhöht, wenn auch hier nichts anderes vorgegeben ist. Somit ist im Beispiel der Wert der Aufzählungskonstante `rk` gleich 0, `evang` gleich 1, `islam` gleich 2, `bud` gleich 3 und `sonstige` gleich dem Wert 4. Verwenden Sie `enum` hingegen wie folgt:

```
enum religion { rk=1, evang, islam, bud=24, sonstige };
```

dann hat die Aufzählungskonstante `rk` den Wert 1, `evang` ist gleich 2, und `islam` ist 3. Der Wert für `bud` lautet jetzt 24, und für `sonstige` erhöht sich der Wert ebenfalls wieder um 1 vom Vorgänger. Er ist somit automatisch 25.

Es spricht nichts dagegen, in einer enum mehrere Konstanten mit demselben Wert zu definieren. Hier ein Beispiel:

```
enum zustand { READ, WRITE, LESEN=0, SCHREIBEN=1 };
```

Hier haben READ und LESEN jeweils den Wert 0, WRITE und SCHREIBEN den Wert 1.

Sicher fragen Sie sich jetzt, welchen Vorteil es hat den Aufzählungstyp enum:

```
enum wochentag{ MO=1, DI, MI, DO, FR, SA, SO };
```

statt define zu verwenden:

```
#define MO 1
#define DI 2
#define MI 3
#define DO 4
...
```

Beide Varianten lassen sich äquivalent in einem Programm verwenden. Der Vorteil der enum-Variante liegt darin, dass enum-Werte im Debugger symbolisch dargestellt werden können, vorausgesetzt, Sie haben einen Bezeichner bzw. Namen für die enum-Aufzählung verwendet.

enum kann innerhalb eines Geltungsbereichs als Deklaration wie folgt verwendet werden:

```
enum wochentag tag1 = MO, tag2 = FR;
// Oder in einer Funktion:
void setDate( enum wochentag tag );
```

12.4 Eigene Typen mit typedef

Bei den Strukturen wurde bereits kurz gezeigt, wie Sie aus komplexeren Namen bzw. Typen etwas einfachere und vor allem übersichtliche Synonyme erstellen können. Ein mit typedef erstelltes Synonym können Sie nach wie vor mit dem umständlicheren Namen verwenden. Hier ein Beispiel:

```
typedef unsigned char Byte_t;
Byte_t byte;     // gleich wie: unsigned char byte;
Byte_t *bytePtr; // gleich wie: unsigned char *bytePtr;
```

Mithilfe von `typedef` haben Sie ein Synonym für `unsigned char` erstellt. Anstatt `unsigned char` können Sie jetzt beispielsweise auch eine Variable mit dem `typedef`-Namen `Byte_t` deklarieren. Im Geltungsbereich der `typedef`-Deklaration ist `Byte_t` damit gleichbedeutend mit dem Typ `unsigned char`. Weitere gängige Beispiele sind:

```
typedef unsigned char* BytePtr_t;
typedef int Word;
```

Auch die Standardbibliothek macht regen Gebrauch von `typedef`, damit sie möglichst portabel bleibt. So könnte in der Headerdatei `<time.h>` der Typ `time_t` wie folgt definiert sein:

```
typedef int time_t;
```

Der Standard schreibt hier nicht vor, wie `time_t` implementiert ist. Bei Systemen, die eine vorzeichenbehaftete 32-Bit-Ganzzahl für `time_t` verwenden, ist daher am 19. Januar 2038 Schluss, weil dann die maximale Anzahl der abgelaufenen Zeit seit dem 1. Januar 1970 mit 2.147.483.647 Sekunden überschritten und mit –2.147.483.648 Sekunden zum 13. Dezember 1901 wird.

Auf einem anderen System könnte die Deklaration auch folgendermaßen aussehen:

```
typedef long long time_t;
```

Wenn hier also `time_t` mit einem 64-Bit-Ganzzahltyp implementiert ist, dann wird unser Programm noch die nächsten 292 Milliarden Jahre zuverlässig arbeiten.

Der Vorteil ist offensichtlich: Bei der Verwendung von `time_t` müssen Sie sich über die Portierung auf andere Systeme keine Gedanken machen. Sie verwenden einfach `time_t`, und den Rest übernimmt die Standardbibliothek für Sie.

typedef kann auch für komplexere Datentypen wie Arrays verwendet werden. Hier ein Beispiel:

```
typedef int Wochen[52];
...
// int-Array mit 52 Elementen
Wochen jahr2010;
// Zugriff wie gewöhnlich:
jahr2010[0] = 1234;
jahr2010[1] = 2345;
...
// Es geht noch komplexer:
Wochen jahr2010_Abteilung[4]; // = jahr2010_Abteilung[52][4]
Wochen *jahrPtr;              // = (*jahrPtr)[52]
```

Hier muss hinzugefügt werden, dass mit typedef das Programm nicht etwa besser oder schneller wird. Vielmehr ist es eine Hilfe, komplexe Speicherobjekte mit einem einfacheren Synonym bei der Entwicklung zu verwenden. Auch bei der Portierung auf andere Systeme kann typedef das Leben unter Umständen erheblich vereinfachen.

12.5 Kontrollfragen und Aufgaben

1. Erklären Sie kurz und bündig, was Strukturen sind.
2. Was sind Unionen, und worin unterscheiden sie sich von den Strukturen?
3. Warum lässt sich im folgenden Codeausschnitt die Strukturvariable artikel1 in der Zeile **(06)** nicht erstellen? Was können Sie tun, damit Sie Zeile **(06)** trotzdem so verwenden können?

```
01  struct Artikel {
02    char schlagzeile[255];
03    int seite;
04    int ausgabe;
05  };
...
06 Artikel artikel1;   // Geht nicht
```

4. Das folgende Listing enthält einige klassische Zugriffsfehler auf Strukturelemente. Ermitteln Sie diese, und bringen Sie das Programm zum Laufen.

```
00 // kap012/aufgabe001.c
01 #include <stdio.h>
02 #include <stdlib.h>
03 #include <string.h>

04 typedef struct artikel{
05   char schlagzeile[255];
06   int seite;
07   int ausgabe;
08 } Artikel;

09 void output( Artikel *a ) {
10   printf("%s\n", a->schlagzeile);
11   printf("%d\n", a->seite);
12   printf("%d\n\n", a->ausgabe);
13 }

14 int main(void) {
15   Artikel art1 = {244, "Die Schlagzeile schlechthin", 33};
16   Artikel *art2;
17   Artikel artArr[2];

18   strncpy( art2->schlagzeile, "Eine Schlagzeile", 255);
19   art2->seite = 212;
20   art2->ausgabe = 43;

21   strncpy( artArr.schlagzeile[0], "Noch eine", 255);
22   artArr.seite[0] = 266;
23   artArr.ausgabe[0] = 67;

24   output( &art1 );
25   output( art2 );
```

```
26    output( &artArr[0] );
27    return EXIT_SUCCESS;
28  }
```

5. Da es keinen direkten und portablen Weg gibt, zwei Strukturen mit dem ==-Operator zu vergleichen, erstellen Sie eine Funktion, die zwei Strukturvariablen auf Gleichheit hin überprüft. Folgende Struktur und folgende Strukturvariablen seien hierfür gegeben:

```
typedef struct artikel {
  char schlagzeile[255];
  int seite;
  int ausgabe;
} Artikel;

Artikel art1 = { "Die Schlagzeile schlechthin", 244, 33 };
Artikel art2 = { "Die Schlagzeile schlechthin", 244, 33 };
Artikel art3 = { "Die-Schlagzeile-schlechthin", 244, 33 };
```

Kapitel 13

Dynamische Datenstrukturen

Mit den Kenntnissen zur dynamischen Speicherverwaltung (Kapitel 11) und den Strukturen (Abschnitt 12.1) haben Sie die Grundlagen für dynamische Datenstrukturen wie verkettete Listen oder binäre Bäume geschaffen. Der Vorteil solcher Listen ist, dass damit wesentlich schneller Elemente eingefügt oder nicht mehr benötigte Elemente gelöscht werden können. Außerdem wird immer nur so viel Speicher verwendet, wie Daten in der Liste vorhanden sind und benötigt werden. Auch das Sortieren ist mit Listen wesentlich schneller zu realisieren als beispielsweise mit Arrays. Muss dann auch noch die Suche extra schnell sein, greift man zu den binären Bäumen, die im Grunde nur eine andere Form von Listen darstellen.

13.1 Verkettete Liste

Verkettete Listen sind zunächst auch nichts anderes als gewöhnliche Strukturtypen mit einem Zeiger als Komponente vom selben Typ wie der Strukturtyp. Damit kann der Zeiger dieser Komponente innerhalb der Struktur die Adresse einer anderen Strukturvariablen vom selben Typ speichern. Da jede Struktur einen solchen Strukturzeiger enthält, können Sie eine Struktur nach der anderen aneinanderhängen. Ein Beispiel hierzu:

```
struct Id3_tag {
  char titel[MAX];
  char kuenstler[MAX];
  char album[MAX];
  short jahr;
  char kommentar[MAX];
```

```
  char genre[MAX];
  struct Id3_tag *next;
};
```

Das Besondere an diesem Zeiger ist, dass er eine Adresse beinhaltet, die einen Wert von demselben Typ wie die Struktur selbst referenziert (`struct Id3_tag`). Mit diesem Zeiger können somit einzelne Strukturvariablen miteinander verkettet werden. Der `next`-Zeiger verweist immer auf die Adresse des nächsten Elements, das wiederum eine Strukturvariable mit denselben Komponenten und ebenfalls einen weiteren Zeiger beinhaltet. Zudem wird noch ein Ende für die Kette benötigt. Dazu kann man dem letzten `next`-Zeiger beispielsweise den `NULL`-Zeiger zuweisen oder einen speziellen Dummy-Ende-Zeiger vom selben Strukturtyp dafür anlegen. Auf jeden Fall sollten Sie einen speziellen Zeiger verwenden, der immer auf das erste Element in der verketteten Liste verweist, um nicht den »Faden« bzw. den Anfang der Kette zu verlieren.

Sie sollten wissen, dass für jedes neue Element in der Liste, das Sie hinzufügen, auch ein Speicherplatz vorhanden sein muss. In der Praxis wird für jedes neue Element mittels `malloc()` Speicher zur Laufzeit angefordert. Beim Löschen eines Elements in der Liste müssen Sie darauf achten, dass Sie den Speicher wieder an das System zurückgeben und die Liste nicht abreißen lassen. Achten Sie ganz besonders darauf, dass die Zeiger richtig miteinander »verkettet« sind. Nichts ist schlimmer als ein ins Nirwana verweisender Zeiger bei einer verketteten Liste mit vielen Daten. Meistens können Sie dann nicht mehr auf diese Daten zugreifen.

Jedes Element in der Liste wird als **Knoten** (engl. *node*) bezeichnet. Ein solcher Knoten enthält die eigentlichen Daten und einen Zeiger auf seinen Nachfolger.

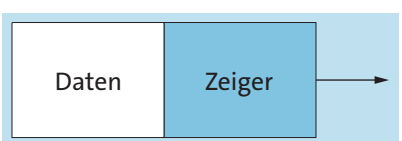

Abbildung 13.1 Ein Listenelement mit seinen Daten und einem Zeiger auf seinen Nachfolger wird als Knoten bezeichnet.

Vereinfachtes Beispiel im Buch

Damit der Buchumfang wegen der Listings zu den verketteten Listen nicht ins Unermessliche wächst, wurde eine sehr einfache Datenstruktur mit nur einer Ganzzahl und dem Zeiger auf das nächste Element zur Demonstration verwendet.

Nachfolgend sehen Sie ein etwas umfangreicheres Beispiel, das aber auf die nötigsten Funktionen beschränkt wurde. Die einzelnen Funktionen werden anschließend noch genauer erläutert.

```
00 // kap013/listing001.c
01 #include <stdio.h>
02 #include <stdlib.h>

03 void dump_buffer(FILE *fp) {
04   int ch;
05   while( (ch = fgetc(fp)) != EOF && ch != '\n' )
06   /* Kein Anweisungsblock nötig */ ;
07 }

08 struct knoten {
     int wert;
     struct knoten *next;
   };

09 typedef struct knoten Knoten_t;
10 typedef struct knoten* KnotenPtr_t;
11 KnotenPtr_t anfang = NULL;

12 void einfuegenKnoten( KnotenPtr_t neu ) {
13   KnotenPtr_t hilfZeiger;
14   if( anfang == NULL ) {
15     anfang = neu;
16     neu->next = NULL;
17   }
18   else {
```

```
19      hilfZeiger = anfang;
20      while(hilfZeiger->next != NULL) {
21        hilfZeiger = hilfZeiger->next;
22      }
23      hilfZeiger->next = neu;
24      neu->next = NULL;
25    }
26  }

27  void neuerKnoten( void ) {
28    KnotenPtr_t neu = malloc(sizeof(Knoten_t));
29    if( neu == NULL ) {
30      printf("Kein Speicher vorhanden!?\n");
31      return;
32    }
33    printf("Wert neuen fuer Knoten eingeben: ");
34    if( scanf("%d", &neu->wert) != 1 ) {
35      dump_buffer(stdin);
36      printf("Fehler bei der Eingabe\n");
37      free(neu); // Speicher wieder freigeben
38      return;
39    }
40    dump_buffer(stdin);
41    einfuegenKnoten( neu );
42  }

43  void loescheKnoten( int val ) {
44    KnotenPtr_t hilfZeiger1;
45    KnotenPtr_t hilfZeiger2;
46    if( anfang != NULL ) {
47      if( anfang->wert == val ) {
48        hilfZeiger1 = anfang->next;
49        free(anfang);
50        anfang = hilfZeiger1;
51      }
52      else {
53        hilfZeiger1 = anfang;
```

```
54        while( hilfZeiger1->next != NULL ) {
55          hilfZeiger2 = hilfZeiger1->next;
56          if( hilfZeiger2->wert == val ) {
57            hilfZeiger1->next = hilfZeiger2->next;
58            free(hilfZeiger2);
59            break;
60          }
61          hilfZeiger1 = hilfZeiger2;
62        }  // Ende while
63      }  // Ende else
64    }  // Ende if
65  }

66  void knotenAuflisten( void ) {
67    KnotenPtr_t hilfZeiger = anfang;
68    printf("Elemente in der Liste:\n");
69    while( hilfZeiger != NULL ) {
70      printf("\t -> %d\n", hilfZeiger->wert);
71      hilfZeiger = hilfZeiger->next;
72    }
73  }

74  int main(void) {
75    int wahl = 0, val = 0;
76    do {
77      printf(" -1- Neues Element hinzufuegen\n");
78      printf(" -2- Element loeschen\n");
79      printf(" -3- Alle Elemente auflisten\n");
80      printf(" -4- Programmende\n");
90      printf(" Ihre Auswahl : ");
91      if( scanf("%d", &wahl) != 1 ) {
92        printf("Fehlerhafte Auswahl\n");
93        wahl = 0;
94        dump_buffer(stdin);
95      }
96      switch( wahl ) {
97        case 1 : neuerKnoten(); break;
```

```
98        case 2 : if( anfang == NULL ) {
99                   printf("Liste ist leer!\n");
101                }
102                else {
103                  printf("Wert zum Loeschen : ");
104                  if( scanf("%d", &val) != 1 ) {
105                    printf("Fehler bei der Eingabe\n");
106                  }
107                  else {
108                    loescheKnoten( val );
109                  }
110                }
111                break;
112       case 3 : if( anfang == NULL ) {
113                  printf("Liste ist leer!\n");
114                }
115                else {
116                  knotenAuflisten();
117                }
118                break;
119     }
120   }while( wahl != 4 );
121   return EXIT_SUCCESS;
122  }
```

Das Programm bei der Ausführung:

```
 -1- Neues Element hinzufügen
 -2- Element löschen
 -3- Alle Elemente auflisten
 -4- Programmende
 Ihre Auswahl : 1
Wert für Knoten eingeben: 6789
 -1- Neues Element hinzufügen
 -2- Element löschen
 -3- Alle Elemente auflisten
 -4- Programmende
```

```
 Ihre Auswahl : 3
1234
2345
4567
6789
 -1- Neues Element hinzufügen
 -2- Element löschen
 -3- Alle Elemente auflisten
 -4- Programmende
 Ihre Auswahl : 2
Wert zum Löschen : 2345
 -1- Neues Element hinzufügen
 -2- Element löschen
 -3- Alle Elemente auflisten
 -4- Programmende
 Ihre Auswahl : 3
1234
4567
6789
```

13.1.1 Neues Element in die Liste einfügen

Die grundlegende Operation auf verketteten Listen dürfte das Hinzufügen neuer Elemente sein. In unserem Listing wird diese Operation mit den Zeilen **(12)** bis **(42)** durchgeführt:

```
12  void einfuegenKnoten( KnotenPtr_t neu ) {
13    KnotenPtr_t hilfZeiger;
14    if( anfang == NULL ) {
15      anfang = neu;
16      neu->next = NULL;
17    }
18    else {
19      hilfZeiger = anfang;
20      while(hilfZeiger->next != NULL) {
21        hilfZeiger = hilfZeiger->next;
22      }
```

```
23      hilfZeiger->next = neu;
24      neu->next = NULL;
25    }
26  }

27  void neuerKnoten( void ) {
28    KnotenPtr_t neu = malloc(sizeof(Knoten_t));
29    if( neu == NULL ) {
30      printf("Kein Speicher vorhanden!?\n");
31      return;
32    }
33    printf("Neuen Wert fuer Knoten eingeben: ");
34    if( scanf("%d", &neu->wert) != 1 ) {
35      dump_buffer(stdin);
36      printf("Fehler bei der Eingabe\n");
37      free(neu); // Speicher wieder freigeben
38      return;
39    }
40    dump_buffer(stdin);
41    einfuegenKnoten( neu );
42  }
```

Legen Sie zuerst einen neuen Knoten dynamisch zur Laufzeit an. Den neuen Knoten legen wir im Listing in den Zeilen **(27)** bis **(42)** an. In Zeile **(28)** wird ein Speicher vom Heap für das neue Element angefordert, und anschließend werden auch gleich die Werte, in unserem Fall nur ein Wert, für das Strukturelement eingelesen. In Zeile **(41)** soll der fertige Knoten zur Liste hinzugefügt werden. Hierbei wird die Adresse des Knotens an die Funktion `einfuegenKnoten()` übergeben, die in den Zeilen **(12)** bis **(26)** aufgeführt ist.

In Zeile **(14)** wird zunächst überprüft, ob überhaupt ein Element in der Liste vorhanden ist. Der Strukturzeiger `anfang` wurde am Anfang des Programms mit dem `NULL`-Zeiger belegt. Hat `anfang` immer noch den `NULL`-Zeiger, handelt es sich um den ersten Knoten in der Liste, der hinzugefügt werden soll. Daher bekommt der Zeiger `anfang` die Adresse des neuen

Knotens (Zeile **(15)**), und der Komponenten-Zeiger des Knotens auf das nächste Element in der Liste bekommt jetzt den `NULL`-Zeiger (Zeile **(16)**).

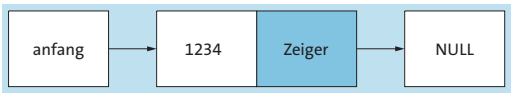

Abbildung 13.2 Der erste Knoten mit dem Wert 1234 wurde hier zu dem Beispiel in der Liste hinzugefügt.

Handelt es sich nicht um das erste Element, das hinzugefügt werden soll, durchlaufen Sie in den Zeilen **(18)** bis **(25)** die komplette verkettete Liste, bis Sie am Ende angekommen sind. Genau genommen wird dies in der `while`-Schleife in den Zeilen **(20)** bis **(22)** gemacht. Ein `hilfZeiger` wird so lange auf das nächste Element gesetzt, bis dieses ein `NULL`-Zeiger ist. Erst dann bekommt das nächste Element, das zuvor noch den `NULL`-Zeiger enthielt, in Zeile **(23)** die Adresse des neuen Knotens. Der `next`-Zeiger des neuen Knotens bekommt anschließend in der Zeile **(24)** den `NULL`-Zeiger.

Tipp: Zeiger auf den letzten Knoten

An dieser Stelle muss angemerkt werden, dass ein Zeiger auf den letzten Knoten erheblich sinnvoller wäre, wenn Sie bei einer verketteten Liste das Element immer nur hinten anhängen. Damit ersparen Sie sich das ständige Durchlaufen aller Knoten. Allerdings hilft Ihnen das Durchlaufen der einzelnen Knoten zum besseren Verständnis, wenn Sie am Ende des Kapitels bei den Aufgaben den neuen Knoten sortiert eingeben sollen. Alternativ können neue Elemente auch sofort immer am Anfang der Liste eingefügt werden. Das minimiert den Zeigeraufwand noch mehr.

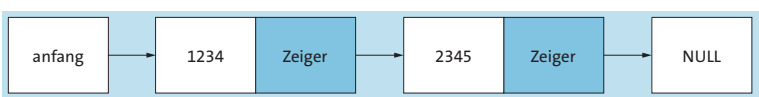

Abbildung 13.3 Ein weiteres Element (hier mit dem Wert 2345) wurde zur verketteten Liste hinzugefügt.

13.1.2 Element ausgeben (und suchen)

Das Ausgeben aller Elemente ist ein Kinderspiel. Hierbei müssen Sie lediglich alle Knoten vom ersten bis zum letzten durchlaufen. Die Funktion wurde mit den Zeilen **(66)** bis **(73)** definiert:

```
66  void knotenAuflisten( void ) {
67    KnotenPtr_t hilfZeiger = anfang;
68    printf("Elemente in der Liste:\n");
69    while( hilfZeiger != NULL ) {
70      printf("\t -> %d\n", hilfZeiger->wert);
71      hilfZeiger = hilfZeiger->next;
72    }
73  }
```

Wichtig ist dabei, dass Sie einen Hilfszeiger verwenden und die einzelnen Knoten nicht über den `anfang`-Zeiger durchlaufen. Denn dann wäre die Liste für immer verloren. Der Hilfszeiger durchläuft in einer Schleife Knoten für Knoten, bis er auf den `NULL`-Zeiger und somit auf das Ende der Liste stößt. Ähnlich wie diese Funktion ist auch die Suchfunktion aufgebaut, die Sie am Ende des Kapitels als Aufgabe erstellen sollen. Hierbei müssen Sie zusätzlich noch Knoten für Knoten überprüfen, ob das gesuchte Element gefunden wurde.

13.1.3 Element aus der Liste entfernen

Das Löschen eines Knotens ist etwas schwieriger. Die Funktion wurde in den Zeilen **(43)** bis **(65)** definiert. Dort wurde der zu löschende Wert als Funktionsparameter mit übergeben:

```
43  void loescheKnoten( int val ) {
44    KnotenPtr_t hilfZeiger1;
45    KnotenPtr_t hilfZeiger2;
46    if( anfang != NULL ) {
47      if( anfang->wert == val ) {
48        hilfZeiger1 = anfang->next;
49        free(anfang);
```

```
50        anfang = hilfZeiger1;
51      }
52      else {
53        hilfZeiger1 = anfang;
54        while( hilfZeiger1->next != NULL ) {
55          hilfZeiger2 = hilfZeiger1->next;
56          if( hilfZeiger2->wert == val ) {
57            hilfZeiger1->next = hilfZeiger2->next;
58            free(hilfZeiger2);
59            break;
60          }
61          hilfZeiger1 = hilfZeiger2;
62        }  // Ende while
63      }  // Ende else
64    }  // Ende if
65  }
```

Überprüfen Sie zunächst in Zeile **(46)**, ob überhaupt ein Knoten in der Liste vorhanden ist. Ist dem nicht so, macht diese Funktion nichts, weil der nachfolgende Anweisungsblock nicht ausgeführt wird.

Erstes Element in der Liste löschen

Überprüfen Sie nun in Zeile **(47)**, ob es der erste Knoten in der Liste ist, der das gesuchte Element enthält und der gelöscht werden soll. Dieser Fall wird in den Zeilen **(48)** bis **(50)** behandelt. `hilfZeiger1` bekommt die Adresse des nächsten (!) Elements hinter dem Element, auf das `anfang` verweist (Zeile **(48)**):

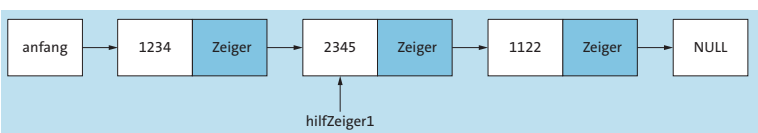

Abbildung 13.4 Das erste Element ist das gesuchte, daher verweist »hilfZeiger1« zunächst auf das nachfolgende Element.

Mit dem `hilfZeiger1` haben Sie den künftigen neuen Anfang der Liste gesichert. Nun können Sie den alten `anfang` mit `free()` in Zeile **(49)** freigeben:

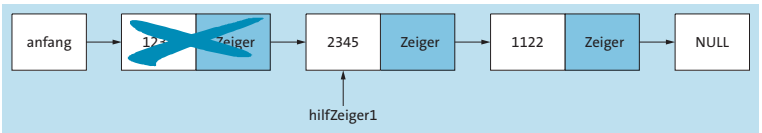

Abbildung 13.5 Der Speicher für das erste Element wurde freigegeben.

Zum Schluss müssen Sie dem Zeiger `anfang` die Adresse des neuen Anfangs in der Liste übergeben, den Sie zuvor in Zeile **(48)** gesichert haben. In Zeile **(50)** bekommt der Zeiger `anfang` die Adresse von `hilfZeiger1`, und die Löschung des ersten Elements ist komplett:

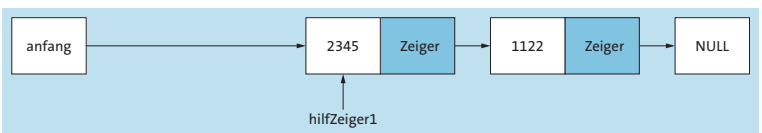

Abbildung 13.6 Der Zeiger »anfang« verweist auf das neue erste Element in der Liste.

Beliebiges Element in der Liste löschen

Sie können aber auch ein beliebiges Element in der Liste löschen, sofern es nicht das erste ist. Dies wird im Code in den Zeilen **(52)** bis **(63)** ausgeführt. Auch hier lassen Sie `hilfZeiger1` zunächst in Zeile **(53)** auf den Anfang der Liste verweisen. Anschließend wird der Code zwischen den Zeilen **(54)** bis **(62)** ausgeführt. Die `while`-Schleife der Zeile **(54)** wird so lange durchlaufen, bis das nächste Element, auf das `hilfZeiger1` verweist, der `NULL`-Zeiger und somit das Ende der Liste ist. Das bedeutet, dass das Element nicht in der Liste vorhanden ist. In der Schleife selbst bekommt `hilfZeiger2` in Zeile **(55)** immer die Adresse des nächsten Elements von `hilfsZeiger1`. Schlägt die Überprüfung der Zeile **(56)** fehl, wurde der von uns gesuchte Knoten nicht gefunden, und `hilfZeiger1` bekommt die Adresse von `hilfZeiger2`. Dann beginnt der nächste Schleifendurchgang.

Im Beispiel wird jetzt davon ausgegangen, dass das zweite Element in der Liste das gesuchte ist – dass Zeile **(56)** also wahrheitsgemäß zutrifft. Grafisch würde die Liste bis zu Zeile **(56)** wie in Abbildung 13.7 aussehen.

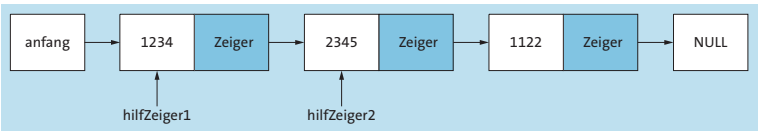

Abbildung 13.7 »hilfZeiger2« verweist auf den von uns gesuchten Knoten.

In Zeile **(57)** wird das zu löschende Element »ausgehängt«, indem der `next`-Zeiger von Knoten `hilfsZeiger1` auf den nächsten Knoten hinter `hilfsZeiger2` verweist. Grafisch dürfte dies einfacher verstehen zu sein (siehe Abbildung 13.8).

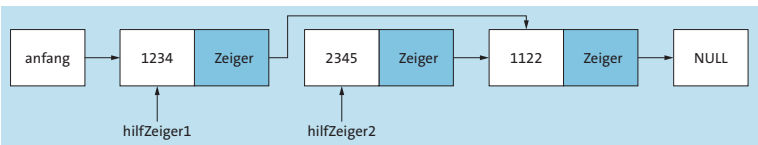

Abbildung 13.8 Das zu löschende Element wird ausgehängt.

Der Knoten ist nun aus der Liste entfernt worden. Jetzt müssen Sie nur noch den reservierten Speicherplatz für den Knoten in Zeile **(58)** mittels `free()` freigeben, damit hier keine Speicherlecks entstehen. Mit `break` werden die Schleife und damit auch die Funktion abgebrochen:

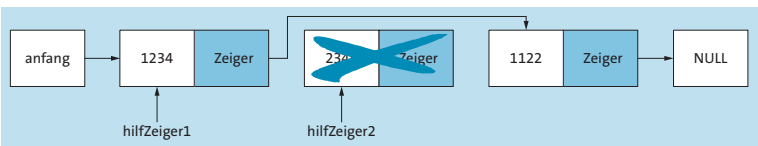

Abbildung 13.9 Um Speicherlecks (Memory Leaks) zu vermeiden, muss der nicht mehr benötigte Knoten wieder freigegeben werden.

13.2 Doppelt verkettete Listen

Benötigen Sie eine Liste, die aus beiden Richtungen durchlaufen werden kann, also von vorne nach hinten und umgekehrt, dann können Sie eine doppelt verkettete Liste verwenden. Im Vergleich zu einer einfach verketteten Liste ändert sich hierbei nur, dass im Strukturtyp ein weiterer Zeiger auf das vorherige Element hinzugefügt werden muss. Natürlich müssen Sie diesen weiteren Zeiger auch zusätzlich im Code verwalten, was den Aufwand bei der Programmierung etwas erhöht. Bezogen auf das Listing in diesem Kapitel sieht die Struktur dann wie folgt aus:

```
struct knoten {
  int wert;
  struct knoten *next;     // Zeiger auf nächstes Element
  struct knoten *previous; // Zeiger auf vorheriges Element
};
```

Durch den zusätzlichen Zeiger wird zwar ein Zeiger mehr und somit auch mehr Speicherplatz für einen Zeiger benötigt. Aber dies kann, richtig implementiert, durch ein effizienteres Sortieren, schnelleres Auffinden, schnelleres Löschen und Einfügen von Elementen in der Liste wieder wettgemacht werden.

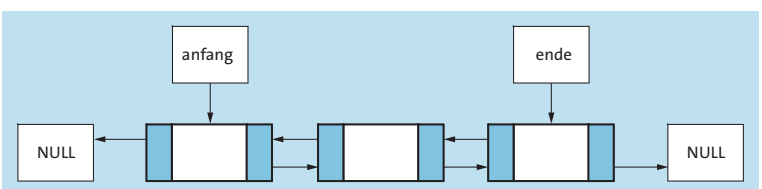

Abbildung 13.10 Eine doppelt verkettete Liste

Dies sind natürlich nicht die einzigen Formen von verketteten Listen. Es können daraus beispielsweise Stacks nach dem LIFO-Prinzip (Last-In-First-Out) oder Queues (Warteschlangen) nach dem FIFO-Prinzip (First-In-First-Out) als abstraktere Datenstrukturen erstellt werden. Auch binäre Suchbäume werden zunächst nur als Struktur mit zwei Zeigern auf dem linken und dem rechten Ast implementiert. Die Implementierung ist dann aller-

dings etwas komplexer als bei verketteten Listen. Hier gibt es noch viele weitere spezielle Formen, die alle die verketteten Listen als Grundlage haben. Darauf soll hier allerdings nicht weiter eingegangen werden.

13.3 Kontrollfragen und Aufgaben

1. Was sind verkettete Listen?
2. Welchen Vorteil haben verkettete Listen gegenüber Arrays?
3. Was sind doppelt verkettete Listen?
4. Welcher Fehler wurde hier beim Löschen des Knotens gemacht?

```
01  void loescheKnoten( int val ) {
02    KnotenPtr_t hilfZeiger1;
03    KnotenPtr_t hilfZeiger2;
04    if( anfang != NULL ) {
05      if( anfang->wert == val ) {
06        hilfZeiger1 = anfang->next;
07        free(anfang);
08        anfang = hilfZeiger1;
09      }
10      else {
11        hilfZeiger1 = anfang;
12        while( hilfZeiger1->next != NULL ) {
13          hilfZeiger2 = hilfZeiger1->next;
14          if( hilfZeiger2->wert == val ) {
15            hilfZeiger1 = hilfZeiger2->next;
16            free(hilfZeiger2);
17            break;
18          }
19          hilfZeiger1 = hilfZeiger2;
20        }  // Ende while
21      }  // Ende else
22    }  // Ende if
23  }
```

5. Ändern Sie die Funktion `einfuegenKnoten()` aus Listing `listing001.c` ab, damit die neuen Elemente in der Liste sortiert eingefügt werden. Verhindern Sie außerdem, dass doppelte Einträge eingefügt werden. Ein Tipp, wie Sie vorgehen können:
   - Überprüfen Sie, ob überhaupt etwas in der Liste vorhanden ist, und fügen Sie das erste Element ein.
   - Jetzt können Sie die einzelnen Knoten durchlaufen und prüfen, ob der aktuelle Wert größer oder kleiner ist als der neu hinzuzufügende Wert (die Reihenfolge entscheiden Sie selbst). Haben Sie den passenden Knoten gefunden, müssen Sie das Element ...
   - ... am Ende einfügen (wenn `NULL` erreicht).
   - ... auf doppelte Werte prüfen und nicht einfügen.
   - ... am Anfang einfügen (wenn nicht weiter als bis zum Anfang iteriert wurde).
   - ... irgendwo dazwischen einfügen.
6. Erweitern Sie das Programm *listing001.c* um eine Funktion zum Suchen eines Knotens mit einem bestimmten Wert in der Liste.

Kapitel 14

Eingabe- und Ausgabe-Funktionen

Ist die Rede von der Ein- und Ausgabe in Programmen, sind damit einerseits die üblichen Dinge wie die Ausgabe von Daten in eine Datei, auf einen Bildschirm oder Drucker gemeint, auf der anderen Seite geht es natürlich um die Möglichkeit, Daten aus einer Datei oder von der Tastatur lesen zu können. Für die Ein- und Ausgabe stellt die Standardbibliothek von C die nötigsten Funktionen zur Verfügung. Sie sind alle in der Headerdatei `<stdio.h>` deklariert.

14.1 Verschiedene Streams und Standard-Streams

Die Ein- und Ausgabe von Daten in C wird über das Konzept der Streams (*data stream* = Datenstrom) realisiert. Beim Öffnen einer Datei beispielsweise wird gewöhnlich ein neuer Datei-Stream (bzw. File-Pointer oder Dateizeiger) angelegt und beim Schließen wieder entfernt. Die einzelnen Streams werden als Ressource von dem Betriebssystem verwaltet, auf dem das Programm ausgeführt wird.

Grafische Oberfläche als Ausgabe in C

Der C-Standard definiert keinen Standard für grafische Oberflächen zur Interaktion mit dem Anwender. Natürlich gibt es jenseits des C-Standards auch viele andere Programmierschnittstellen in C für Anwendungsprogramme und Betriebssystem-APIs, die Sie für die Programmierung verwenden können. Allerdings sind dies meistens Bibliotheken, die vom Compiler oder Betriebssystem abhängig sind. Darauf wird in diesem Buch nicht eingegangen.

14.1.1 Stream im Textmodus

Ein Stream im Textmodus liest und schreibt einzelne Zeichen eines Textes. Gewöhnlich wird dieser Text in einzelne Zeilen aufgeteilt. Bei solchen Streams werden alle sichtbaren Zeichen und einige Steuercodes, etwa Zeilenschaltung oder Tabulatoren, verwendet. Da bei Windows-Systemen das Zeilenende oft mit `\r\n` ausgegeben wird und Linux/Unix-Systeme dafür nur `\n` verwenden, führt der Compiler hier eine automatische Konvertierung durch. Um diese müssen Sie sich aber nicht kümmern. Das Ende eines Textes wird gewöhnlich durch das Steuerzeichen `^Z` (Zeichencode 26) angezeigt (das auch mit der Tastenkombination Strg+Z unter Windows oder Strg+D unter Linux/UNIX »ausgelöst« werden kann).

14.1.2 Stream im binären Modus

Bei einem Stream im binären Modus wird nicht mehr auf den Inhalt wie einzelne Zeilen, Zeichen oder Sonderzeichen geachtet, sondern diese werden Byte für Byte verarbeitet. Daher stehen Daten, die auf einem bestimmten System in einem binären Modus geschrieben wurden, auf demselben System beim Lesen auch exakt so wieder zur Verfügung. Es werden keinerlei automatische Konvertierungen durchgeführt.

14.1.3 Standard-Streams

Drei Streams, die Standard-Streams, sind bei jedem C-Programm von Anfang an vorhanden. Bei den Standard-Streams handelt es sich um Zeiger auf ein `FILE`-Objekt. Nachfolgend sehen Sie die Standard-Streams:

- `stdin` – Die Standardeingabe (**st**an**d**ard **in**put), die gewöhnlich mit der Tastatur verbunden ist. Der Stream ist zeilenweise gepuffert.
- `stdout` – Die Standardausgabe (**st**an**d**ard **out**put) ist mit dem Bildschirm zur Ausgabe verbunden. Auch die Standardausgabe wird zeilenweise gepuffert.
- `stderr` – Die Standardfehlerausgabe (**st**an**d**ard **err**or output) ist wie `stdout` ebenfalls mit dem Bildschirm verbunden, aber die Ausgabe erfolgt ungepuffert.

Alle drei Standard-Streams können umgelenkt werden. Die Umlenkung kann dabei auch programmtechnisch mit der Standardfunktion `freopen()` durchgeführt werden oder über die Umgebung des Programms, beispielsweise mit einem Umleitungszeichen in der Kommandozeile.

Standardfehlerausgabe

Bei den bisherigen Beispielen haben Sie immer eine Fehlerausgabe mit `printf()` auf die Standardausgabe (`stdout`) gemacht. In der Praxis ist es allerdings empfehlenswerter, diese Ausgabe auf `stderr` auszugeben. Hierfür bietet sich beispielsweise die Funktion `fprintf()` an, mit der Sie den Ausgabe-Stream explizit angeben können:

```
fprintf(stderr, "Fehlermeldung");
```

Der Vorteil, eine Fehlermeldung auf `stderr` anstatt `stdout` auszugeben: Wenn die gewöhnliche Standardausgabe umgelenkt wird, werden die Fehlermeldungen nicht mehr am Bildschirm angezeigt. Ansonsten funktioniert die Funktion `fprintf()` genauso wie schon `printf()`, nur dass Sie damit eben den Stream als erstes Argument vorgeben können.

14.2 Dateien

Häufig wendet man die Dateifunktionen an, ohne sich Gedanken darüber zu machen, was eine Datei eigentlich ist. Im Prinzip können Sie sich eine Datei als ein riesengroßes `char`-Array vorstellen. Das `char`-Array besteht dabei aus einer bestimmten Byte-Folge – unabhängig davon, ob es sich um eine Textdatei oder um eine ausführbare binäre Datei handelt. Erst bei der Verarbeitung der Daten bekommen die einzelnen Bytes oder Zeichen eine Bedeutung.

Öffnen Sie in C eine Datei mit einer der dafür vorgesehenen Standardfunktionen wie `fopen()`, `freopen()` oder `tmpfile()`, wird ein Speicherobjekt vom Typ `FILE` angelegt und initialisiert. Eine erfolgreich geöffnete Datei in C liefert immer einen Zeiger auf ein `FILE`-Speicherobjekt zurück, das mit dem Stream verbunden ist. Ob hierbei ein Text- oder ein binärer Stream

verwendet wird, kann mit einer zusätzlichen Option der Funktionen angegeben werden.

Das FILE-Objekt ist letztendlich auch ein Strukturtyp, der in der Headerdatei <stdio.h> deklariert ist. Diese Struktur enthält alle nötigen Informationen der Komponenten, die Sie für die Ein- und Ausgabe-Funktionen benötigen. Sie beinhaltet unter anderem:

- den Puffer – die Anfangsadresse, den aktuellen Zeiger, die Größe
- den File-Deskriptor (eine Ganzzahl mit der aktuellen Dateiposition der niedrigeren Ebene)
- die Position des Schreib- oder Lesezeigers
- die Fehler- und Dateiende-Flags

Wenn Sie eine Datei erfolgreich geöffnet haben, können Sie mithilfe des zurückgegebenen FILE-Speicherobjekts die Daten des Streams über die Standardfunktionen, die einen Zeiger auf das FILE-Speicherobjekt als Argument erwarten, auslesen und ändern.

14.3 Dateien öffnen

Wollen Sie eine Datei bearbeiten, müssen Sie diese zunächst öffnen. Wie im vorigen Abschnitt erwähnt, stehen Ihnen hierzu drei Funktionen zur Verfügung, die alle einen Zeiger auf ein FILE-Objekt zurückliefern. Hierzu die einzelnen Funktionen im Überblick:

```
#include <stdio.h>
FILE *fopen( const char * restrict filename,
             const char * restrict mode );
```

Mit der Funktion fopen() öffnen Sie eine Datei mit dem Namen filename. filename darf auch eine Pfadangabe sein. Wenn eine Datei nicht geöffnet werden konnte oder nicht existiert, gibt die Funktion den NULL-Zeiger zurück. Mit dem zweiten Argument mode bestimmen Sie den Zugriffsmodus. Bei der Festlegung des Zugriffsmodus ist nur ein String mit einem bestimmten Inhalt erlaubt. Die einzelnen Zugriffsmodi und ihre Bedeutung sind in der Tabelle 14.1 aufgelistet.

Modus	Bedeutung
`"r"`	Öffnet eine Datei zum Lesen (`r`=*read*).
`"w"`	Öffnet eine Datei zum Schreiben. Existiert diese Datei nicht, wird sie neu erzeugt. Existiert die Datei mit Inhalt, wird dieser auf 0 gekürzt und ist somit verloren (`w`=*write*).
`"a"`	Wie der Modus `"w"`, nur wird hierbei der Inhalt einer eventuell existierenden Datei nicht gelöscht, sondern der neu zu schreibende Inhalt wird am Dateiende angefügt (`a`=*append*).
`"r+"`	Öffnet eine Datei zum Lesen und Schreiben. Existiert diese Datei nicht, wird der `NULL`-Zeiger zurückgegeben.
`"w+"`	Wie mit `"r+"` wird eine Datei zum Lesen und Schreiben geöffnet. Es wird ggf. eine neue Datei angelegt, wenn sie nicht existiert oder der alte Inhalt der ursprünglichen Datei gelöscht wird.
`"a+"`	Hiermit wird eine Datei zum Lesen und Schreiben am Ende der Datei geöffnet. Ist die Datei noch nicht vorhanden, wird sie neu angelegt.

Tabelle 14.1 Modus zum Öffnen einer Datei mit »fopen()«

Mithilfe des +-Symbols können Sie immer eine Datei zum Lesen und Schreiben öffnen.

Allerdings muss beim Wechseln vom Schreib- zum Lesezugriff die Schreiboperation mit der Funktion `fflush()` `fsetpos()`, `fseek()` oder `rewind()` abgeschlossen werden, wenn Sie unmittelbar darauf eine Leseoperation ausführen wollen. Wollen Sie nach einer Leseoperation eine Schreiboperation aufrufen, müssen Sie eine der Funktionen `fseek()`, `fsetpos()` oder `rewind()` zur richtigen Positionierung des Schreibzeigers aufrufen, es sei denn, es wurde das Dateiende (`EOF`) gelesen.

Zugriffsrechte

Damit Sie überhaupt eine Datei in einem bestimmten Modus öffnen können, müssen die entsprechenden Zugriffsrechte vorhanden sein. Sind die Rechte für normale Benutzer bei einer Datei nur auf Lesen eingestellt, schlägt der Versuch, die Datei zum Schreiben zu öffnen, fehl. In diesem Fall wird ein `NULL`-Zeiger zurückgegeben. Unter Windows werden diese Zugriffsrechte allerdings weniger streng als auf Linux-/Unix-Systemen behandelt.

An alle Modi können Sie noch das Zeichen `b` anfügen (beispielsweise `"rb"` oder `"w+b"`). Damit wird eine Datei im Binärmodus geöffnet, d. h., sie wird mit einem binären Stream verbunden. Ohne das zusätzliche Zeichen `b` werden alle Dateien als Textdatei, also im Textmodus geöffnet.

Hier ein einfaches Beispiel zur Funktion `fopen()`:

```
00 // kap014/listing001.c
01 #include <stdio.h>
02 #include <stdlib.h>
03 #include <string.h>
04 #define NAME 1024

05 int main(void) {
06   char filename[NAME];
07   printf("Welche Datei soll geoeffnet werden: ");
08   if( fgets(filename, sizeof(filename), stdin) == NULL ) {
09     fprintf(stderr, "Fehler bei der Eingabe\n");
10     return EXIT_FAILURE;
11   }
12   // Newline entfernen
13   size_t p = strlen(filename);
14   filename[p-1] = '\0';
15   FILE *fp = fopen(filename, "r");
16   if( fp != NULL ) {
17     printf("Datei zum Lesen geoeffnet\n");
18     fclose(fp);
```

```
19    }
20    else {
21      fprintf(stderr,"Datei konnte nicht geoeffnet werden");
22    }
23    return EXIT_SUCCESS;
24  }
```

Bei diesem Beispiel sollen Sie eine Datei zum Lesen (gerne auch mit Pfad) eingeben. In Zeile **(15)** wird versucht, diese Datei zum Lesen zu öffnen. Ob das klappt oder nicht, wird in Zeile **(16)** überprüft: Ist der Rückgabewert von `fopen()` ungleich `NULL`? Ist dies der Fall, wird Entsprechendes ausgegeben, und die Datei wird in Zeile **(18)** mit `fclose()` gleich wieder geschlossen.

Exklusiver Dateizugriff

Mit dem C11-Standard wurde zur Funktion `fopen()` ein exklusiver Dateizugriff hinzugefügt, mit dem geprüft wird, ob eine Datei **nicht** vorhanden ist, und mit dem anschließend die Datei erstellt und geöffnet wird. Sollte eine Datei vorhanden sein, schlägt der Funktionsaufruf von `fopen()` fehl. Bisher musste dies mit einem zweimaligen Aufruf von `fopen()` gemacht werden, indem erst mit dem Modus `"r"` getestet wurde, ob die Datei vorhanden ist, um dann eventuell mit dem Modus `"w"` diese Datei zu erzeugen. Da es sich hierbei um zwei Aktionen handelt, wobei zwischen dem ersten und zweite `fopen()`-Aufruf ein Zeitfenster für Angreifer vorhanden ist, in dem diese eine eigene Datei mit demselben Namen erzeugen könnten, war diese Möglichkeit immer schon etwas unsicher.

Alternativen waren dabei die Möglichkeiten in POSIX, die Flags `O_EXCL` und `O_CREATE` zu kombinieren und die Funktion `open()` aufzurufen, oder der exklusive Modus mit *x*, den manche C-Standardbibliotheken hierfür schon anboten. Dieser wurde mit C11 standardisiert und kann daher für den exklusiven Dateizugriff als atomare Aktion aufgerufen werden. Somit können Sie mit `fopen("datei.txt", "wx")` eine Datei erzeugen und haben auch einen exklusiven Zugriff zum Schreiben auf diese Datei. Auch der Modus `"w+x"` kann hierbei verwendet werden, womit Sie dasselbe erreichen, nur dass neben dem Schreibzugriff auch der Lesezugriff exklusiv ist. Zusätzlich können Sie in beiden Modi das `b` anhängen, um die Dateien im binären Modus zu öffnen (beispielsweise: `"wbx"` oder `"wb+x"`).

Weitere Funktionen

Jetzt kommen wir zu den anderen beiden Funktionen, mit denen Sie eine Datei öffnen können. Zunächst die Funktion `freopen()`:

```
#include <stdio.h>
FILE *freopen( const char * restrict filename,
               const char * restrict mode,
               FILE * restrict stream );
```

Die Funktion `freopen()` öffnet wie schon `fopen()` die Datei `filename` im Modus `mode`. Im Gegensatz zu `fopen()` wird allerdings kein neuer Stream erzeugt, sondern es wird der Stream verwendet, den Sie mit dem dritten Argument `stream` verwenden. Diese Funktion wird gerne genutzt, um einen bereits geöffneten Stream zu schließen und in einem neuen Modus wieder zu öffnen.

Eine weitere Anwendung von `freopen()` ist es, die Standard-Streams `stdin`, `stdout` und `stderr` umzulenken. Hierzu ein einfaches Beispiel:

```
00  // kap014/listing002.c
01  #include <stdio.h>
02  #include <stdlib.h>

03  int main(void) {
04    printf("Diesen Text koennen Sie sehen\n");
05    if( freopen("logfile.txt", "w", stdout) == NULL ) {
06      fprintf(stderr, "Fehler bei freopen\n");
07      return EXIT_FAILURE;
08    }
09    printf("Dieser Text steht in logfile.txt\n");
10    return EXIT_SUCCESS;
11  }
```

In Zeile **(04)** wird die `printf`-Ausgabe noch wie gewöhnlich über die Standardausgabe gemacht. Mit Zeile **(05)** lenken Sie über `freopen()` die Standardausgabe (`stdout`) in die Datei `logfile.txt` um. Daher wird der Text von Zeile **(09)** nicht mehr auf den Bildschirm, sondern in die Datei *logfile.txt* geschrieben.

Die letzte Funktion zum Öffnen einer Datei ist `tmpfile()`. Hier die Syntax dazu:

```
#include <stdio.h>
FILE *tmpfile(void);
```

Mit dieser Funktion wird eine neue temporäre Datei mit einem eindeutigen Namen erzeugt. Die Datei wird binär mit dem Modus `"wb+"` zum Lesen und Schreiben geöffnet. Die temporäre Datei wird nach dem Schließen mittels `fclose()` oder mit Beendigung des Programms automatisch wieder gelöscht. Konnte keine temporäre Datei geöffnet werden, wird der `NULL`-Zeiger zurückgegeben. Wenn Sie das Programm abnormal beendet haben, ist es implementierungsabhängig, ob die temporäre Datei gelöscht wird oder nicht. Auch die Anzahl der maximal gleichzeitig geöffneten temporären Dateien ist mit `TMP_MAX` in der Headerdatei `<stdio.h>` beschränkt.

14.4 Dateien schließen

Wenn Sie mit der Datei fertig sind, sollten Sie sie mit der Funktion `fclose()` wieder schließen. Die Syntax dieser Funktion lautet:

```
#include <stdio.h>
int fclose( FILE *stream );
```

Nach dem Aufruf der Funktion werden noch ungeschriebene Daten in den Puffer geschrieben. Wenn ein Fehler beim Schließen einer Datei aufgetreten ist, wird `EOF` zurückgegeben, ansonsten `0`.

Beachten Sie hierbei, dass `fclose()` einen gültigen `FILE`-Zeiger erwartet. Wenn Sie einen ungültigen Zeiger oder `NULL` an `fclose()` übergeben, ist das weitere Verhalten undefiniert.

Zwar wird ein Stream auch nach Beendigung eines Programms geschlossen (das schließt auch eine abnormale Beendigung mit ein), aber nur mithilfe der Funktion `fclose()` ist wirklich sichergestellt, dass noch nicht geschriebene Daten im Puffer in die Datei geschrieben werden.

In der Praxis ist es daher ratsam, eine Datei gleich nach dem Abschluss der Datenverarbeitung zu schließen, um so die Gefahr eines Datenverlusts im Falle eines Programmabsturzes zu reduzieren.

Limit maximal geöffneter Dateien

Ein weiteres wichtiges Argument, eine Datei zu schließen: Sie können keine unbegrenzte Anzahl Dateien auf einmal geöffnet lassen. Der Standard von C schreibt mit `FOPEN_MAX` in `<stdio.h>` vor, wie viele `FILE`-Objekte Sie maximal in einem Programm verwenden können. Laut Standard sollten es mindestens acht offene Streams (ohne die Standard-Streams `stdin`, `stdout` und `stderr`) sein. In der Praxis liegt dieser Wert allerdings meistens viel höher.

14.5 Fehler oder Dateiende prüfen

Wenn Sie jetzt anschließend auf die Streams mit Funktionen operieren und eine Funktion abgebrochen wurde, ist es nicht immer eindeutig, ob ein Fehler aufgetreten ist oder ob einfach nur das Dateiende erreicht wurde. So wird beispielsweise `EOF` von einer Lesefunktion zurückgegeben, wenn das Dateiende erreicht wird. Der Rückgabewert `EOF` kann aber auch bei einem aufgetretenen Fehler dieser Funktion zurückgegeben werden. Damit Sie diese beiden Fälle unterscheiden können, lernen Sie die entsprechenden Funktionen mit `feof()` und `ferror()` kennen. Mit diesen können Sie prüfen, ob das Dateiende erreicht wurde oder ob ein Fehler aufgetreten ist.

End-of-File indicator

Wenn Sie beim Lesen einer Datei das Dateiende erreicht haben, zeigt dies ein spezielles Flag an. Dieses Flag wird als *end-of-file indicator* bezeichnet. Ob das Flag gesetzt ist, können Sie mit der Funktion `feof()` ermitteln. Die Funktion erwartet als Argument den Stream, den Sie überprüfen wollen.

Die Syntax von `feof()`lautet:

```
#include <stdio.h>
int feof(FILE *fp);
```

Wenn das Dateiende erreicht wurde und der *end-of-file indicator* gesetzt ist, gibt die Funktion einen Wert ungleich 0 zurück. Wurde das Dateiende noch nicht erreicht, wird gleich 0 zurückgegeben.

Error indicator

Ein weiteres Flag, das in Verbindung mit den Streams gesetzt sein kann, ist der *error indicator*, mit dem Sie feststellen können, ob ein Fehler während einer Operation auf einen Stream aufgetreten ist. Dieses Flag können Sie mit der Funktion `ferror()` überprüfen.

Die Syntax der Funktion lautet:

```
#include <stdio.h>
int ferror(FILE* fp);
```

Bei einem Fehler im Stream `fp` gibt diese Funktion einen Wert ungleich 0 zurück. 0 wird zurückgegeben, wenn kein Fehler aufgetreten ist.

Informationen zum Fehler ausgeben

Tritt bei einer Funktion ein Fehler auf, wird in der globalen Variablen `errno` ein entsprechender Fehlerwert gesetzt. Mit den beiden Funktionen `strerror()` und `perror()` können Sie diese Fehlermeldung ausgeben lassen. Die Variable `errno` ist in der Headerdatei `<errno.h>` definiert. Hier ein Beispiel:

```
long aktPos = ftell(fp);
if(aktPos < 0) {
   perror("Fehler bei ftell()");
}
```

Ist in diesem Fall die Funktion `ftell()` fehlgeschlagen, gibt die Funktion `perror()` auf dem GCC eine Fehlermeldung aus, etwa:

```
Fehler bei ftell(): Bad file descriptor
```

Der Doppelpunkt und die Fehlermeldung dahinter mit einem Newline-Zeichen wurden von der Funktion `perror()` hinzugefügt. Leider sind die meisten Fehlermeldungen compiler- und systemabhängig. Lediglich

folgende drei Fehlerkonstanten sind in der Headerdatei <errno.h> garantiert und können daher zuverlässig von perror() oder strerror() ausgegeben werden:

Konstante	Bedeutung
EDOM	*Domain error*: unzulässiges Argument für eine mathematische Funktion
EILSEQ	*Illegal sequence*: Bei der Verarbeitung von Multibyte-Zeichen wurde ein Byte entdeckt, das kein gültiges Zeichen darstellt.
ERANGE	*Range error*: Das Ergebnis liegt außerhalb des Bereichs.

Tabelle 14.2 Standardmäßige Fehlerkonstanten

Fehler- und EOF-Flag zurücksetzen – clearerr()

Wollen Sie die Flags des *end-of-file* und *error indicators* eines Streams zurücksetzen, können Sie die Funktion clearerr() aufrufen:

```
#include <stdio.h>
void clearerr(FILE* fp);
```

14.6 Funktionen für die Ein- und Ausgabe

Im folgenden Abschnitt lernen Sie die Funktionen kennen, mit denen Sie einzelne Zeichen, Zeilen oder ganze Datenblöcke an einen Stream übertragen oder aus diesem lesen können.

14.6.1 Einzelne Zeichen lesen

Zum Lesen von einzelnen Zeichen aus einem Stream sind in der Headerdatei <stdio.h> folgende Funktionen vorhanden:

```
int fgetc(FILE* fp);
int getc(FILE* fp);
int getchar();
```

Mit `fgetc()` können Sie das nächsten Zeichen aus dem Stream `fp` als `unsigned char` lesen. Als Rückgabewert erhalten Sie bei Erfolg das gelesene Zeichen als `int`-Wert konvertiert oder bei einem aufgetretenen Fehler `EOF`. Ebenfalls `EOF` wird zurückgegeben, wenn der Stream als nächstes Zeichen auf das Dateiende verweist. Der Unterschied zwischen `fgetc()` und `getc()` besteht darin, dass `getc()` als Makro implementiert sein darf. Mit `getchar()` hingegen lesen Sie ein Zeichen von der Standardeingabe (`stdin`). `getchar()` ist daher gleichwertig mit `fgetc(stdin)`.

Zeichen in den Stream zurückstellen

Haben Sie ein zu viel gelesenes Zeichen aus dem Stream geholt, können Sie es mit der Funktion `ungetc()` wieder in den Stream zurückschieben. Die Syntax dieser Funktion lautet:

```
int ungetc(int c, FILE *fp);
```

Damit schieben Sie das Zeichen `c` (das in ein `unsigned char` konvertiert wird) in den Eingabe-Stream `fp` zurück. Die Funktion gibt das zurückgeschobene Zeichen zurück, oder sie meldet bei einem Fehler `EOF`. Damit ist `c` das erste Zeichen, das bei der nächsten Leseoperation aus dem Stream `fp` erneut gelesen wird. Das gilt allerdings nicht mehr, wenn vor der nächsten Leseoperation eine der Funktionen `fflush()`, `rewind()`, `fseek()` oder `fsetpos()` aufgerufen wurde.

14.6.2 Einzelne Zeichen schreiben

Die Funktionen zum Schreiben von einzelnen Zeichen in den Stream lauten:

```
#include <stdio.h>
int fputc(int c, FILE *fp);
int putc(int c, FILE *fp);
int putchar(int c);
```

Mit `fputc()` schreiben Sie das von `int` in `unsigned char` umgewandelte Zeichen `c` in den verbundenen Stream `fp`. Zurückgegeben wird ein nicht negativer Wert bei Erfolg oder `EOF` im Falle eines Fehlers. Die Version `putc()` entspricht der Funktion `fputc()`. Es handelt sich hierbei aber um ein

Makro. Mit `putchar()` wird das angegebene Zeichen `c` auf die Standardausgabe (`stdout`) geschrieben. `putchar(c)` entspricht somit `fputc(c, stdout)`.

Nachfolgend sehen Sie ein einfaches Beispiel, welches das zeichenweise Lesen und Schreiben demonstrieren soll:

```
00 // kap014/listing003.c
01 #include <stdio.h>
02 #include <stdlib.h>
03 #define FILENAME "listing003.c" // Anpassen
04 #define COPY "listing003.bak"   // Anpassen

05 int main(void) {
06   FILE *fpr = fopen( FILENAME, "r" );
07   if( fpr == NULL ) {
08     fprintf(stderr, "Fehler beim Oeffnen: %s\n", FILENAME);
09     return EXIT_FAILURE;
10   }
11   FILE *fpw = fopen( COPY, "w" );
12   if( fpw == NULL ) {
13     fprintf(stderr, "Fehler beim Oeffnen: %s\n", COPY);
14     return EXIT_FAILURE;
15   }
16   int c;
17   while ( (c=fgetc(fpr)) != EOF ) {
18     if( c > 127 ) {
19       fputc('-', stdout);
20     }
21     else {
22       fputc(c, stdout);
23     }
24     if( fputc(c, fpw ) == EOF ) {
25       fprintf(stderr, "Fehler beim Schreiben\n");
26       break;
27     }
28   }
29   if( c == EOF ) {
```

```
30      if( feof(fpr) ) {
31        printf("Dateiende erreicht\n");
32      }
33      else if( ferror(fpr) ) {
34        printf("Ein Fehler beim Lesen ist aufgetreten!\n");
35      }
37    }
38    fclose(fpr);
39    fclose(fpw);
40    return EXIT_SUCCESS;
41  }
```

In Zeile **(06)** wird eine Datei zum Lesen geöffnet. In Zeile **(11)** wird eine Datei zum Schreiben geöffnet. Ziel des Beispiels ist es, eine Datei zeichenweise bzw. byteorientiert zu kopieren. In Zeile **(17)** lesen Sie so lange Zeichen für Zeichen (bzw. Byte für Byte) aus dem Lese-Stream `fpr` ein, bis Sie auf das Dateiende (`EOF`) stoßen. In Zeile **(18)** überprüfen Sie, ob der Wert des Zeichens größer als 127 war. In diesem Fall bedeutet das, dass dieses Zeichen oberhalb des 7-Bit-ASCII-Zeichensatzes ist. Dort könnten Sie auf einigen Systemen beispielsweise mit Umlauten und anderen speziellen Zeichen Probleme bekommen. Das kommt immer auf den Zeichensatz an, der verwendet wird. Daher wird anstatt eines solchen Zeichens, das vielleicht nicht richtig ausgegeben wird, einfach ein Trennstrich in Zeile **(19)** auf dem Bildschirm ausgegeben. Alle anderen Zeichen unter dem ASCII-Wert 127 werden ganz normal in Zeile **(22)** ausgegeben. Unbehandelt wird allerdings jedes Zeichen auf jeden Fall so in den Schreib-Stream `fpw` in Zeile **(24)** geschrieben, wie es gelesen wurde.

Das Beispiel zeigt eine interessante Möglichkeit auf, wie Sie einen Filter schreiben können. So setzen Sie nicht wie in Zeile **(19)** einen Trennstrich, sondern Sie schreiben eine spezielle Funktion, die sich mit diesem Problem auseinandersetzt.

14.6.3 Zeilenweise einlesen

Zum zeilenweisen Einlesen steht Ihnen folgende bereits wohlbekannte Funktion zur Verfügung:

```
#include <stdio.h>
char *fgets(char *buf, int n, FILE *fp );
```

Die Funktion wurde im Buch schon des Öfteren verwendet. Damit lesen Sie vom Stream `fp` maximal `n-1` Zeichen in den Puffer `buf` und hängen immer ein Stringende-Zeichen am Ende an. Ein Lesevorgang wird beim Erreichen eines Newline-Zeichens (Zeilenende) oder beim Dateiende beendet. Als Rückgabewert gibt diese Funktion entweder die Anfangsadresse von `buf` oder, wenn kein einziges Zeichen eingelesen wurde, den `NULL`-Zeiger zurück.

14.6.4 Zeilenweise schreiben

Die Gegenstücke zum Schreiben eines nullterminierten Strings in einen Stream sind:

```
#include <stdio.h>
int fputs(const char* str, FILE *fp);
int puts(const char* str);
```

Mit `fputs()` schreiben Sie den nullterminierten String, auf den der Zeiger `str` verweist, in den Stream `fp`. Das Nullzeichen `'\0'` wird **nicht** (!) mit in den Stream geschrieben. Die Funktion `puts()` hingegen gibt den String auf den Zeiger `str` auf die Standardausgabe `stdout` auf dem Bildschirm mit einem zusätzlichen *Newline*-Zeichen aus. Der Rückgabewert ist ein nichtnegativer Wert bei Erfolg oder `EOF` im Fall eines Fehlers.

Hierzu ein einfaches Beispiel, das einige gängige Funktionen zum zeilenweise Lesen und Schreiben in der Praxis demonstriert:

```
00  // kap015/listing004.c
01  #include <stdio.h>
02  #include <stdlib.h>
03  #include <string.h>
04  #define LINEBUF 1024

05  void killNL( char *str ) {
06    size_t p = strlen(str);
```

```
07    if(str[p-1] == '\n') {
08      str[p-1] = '\0';
09    }
10  }

11  void dump_buffer(FILE *fp) {
12    int ch;
13    while( (ch = fgetc(fp)) != EOF && ch != '\n' )
14    /* Kein Anweisungsblock nötig */ ;
15  }

16  void countLineOut( FILE *rfp ) {
17    char buf[LINEBUF];
18    int count = 1;
19    while( fgets(buf, LINEBUF, rfp) != NULL ) {
20      printf("%3d | ", count);
21      fputs( buf, stdout );
22      count++;
23    }
24  }

25  void search( FILE *fp ) {
26    char str[LINEBUF], buf[LINEBUF];
27    int count = 1;
28    printf("Wonach wollen Sie suchen: ");
29    if( fgets(str, LINEBUF, stdin) == NULL ) {
30      fprintf(stderr, "Fehler bei der Eingabe\n");
31      return;
32    }
33    killNL(str);
34    while( fgets(buf, LINEBUF, fp) != NULL ) {
35      if(strstr(buf, str) != 0) {
36        printf("%3d  :  %s", count, buf);
37      }
38      count++;
39    }
40  }
```

```
41 void copyFile( FILE *rfp, FILE *wfp ) {
42   char buf[LINEBUF];
43   while( fgets(buf, LINEBUF, rfp ) != NULL ) {
44     if( fputs(buf, wfp) == EOF ) {
45       if( ferror(wfp) ) {
46         fprintf(stderr, "Fehler beim Schreiben\n");
47         return;
48       }
49     }
50   }
51 }

52 int main(void) {
53   char filename1[LINEBUF], filename2[LINEBUF];
54   FILE *wfp = NULL;
55   int input = 0;
56   printf("Datei zum Lesen: ");
57   if( fgets(filename1, LINEBUF, stdin) == NULL ) {
58     fprintf(stderr, "Fehler bei der Eingabe\n");
59     return EXIT_SUCCESS;
60   }
61   killNL(filename1);
62   FILE *rfp = fopen(filename1, "r");
63   if( rfp == NULL ) {
64     fprintf(stderr, "Fehler beim Oeffnen\n");
65     return EXIT_FAILURE;
66   }
67   printf("Was wollen Sie mit der Datei machen?\n");
68   printf("-1- Zeilen zaehlen (Bildschirmausgabe)\n");
69   printf("-2- Zeilen zaehlen (in Datei schreiben)\n");
70   printf("-3- Suchen\n");
71   printf("-4- Kopieren\n");
72   printf("Ihre Auswahl: ");
73   if( scanf("%d", &input) != 1 ) {
74     fprintf(stderr, "Fehler bei der Eingabe\n");
75     return EXIT_FAILURE;
76   }
```

```
77    dump_buffer(stdin);
78    switch( input ) {
79      case 1 :
80      case 2 : if( input == 2 ) {
81                 printf("Dateiname der Kopie: ");
82                 if(fgets(filename2,LINEBUF,stdin)==NULL) {
83                   fprintf(stderr,"Fehler bei der Eingabe");
84                   break;
85                 }
86                 killNL(filename2);
87                 wfp = freopen(filename2, "w", stdout);
88                 if( wfp == NULL ) {
89                   fprintf(stderr, "Fehler bei Oeffnen\n");
90                   break;
91                 }
92               }
93               countLineOut(rfp);
94               break;
95      case 3 : rfp = fopen(filename1, "r");
96               if( rfp != NULL ) {
97                 search(rfp);
98               }
99               break;
100     case 4 : printf("Dateiname der Kopie: ");
101              if(fgets(filename2, LINEBUF, stdin)==NULL) {
102                fprintf(stderr, "Fehler bei der Eingabe\n");
103                break;
104              }
105              killNL(filename2);
106              wfp = fopen(filename2, "w");
107              if( wfp != NULL ) {
108                copyFile( rfp, wfp );
109              }
110              else {
111                fprintf(stderr, "Fehler beim Oeffnen\n");
112              }
113              break;
```

```
114   }
115   if( rfp !=  NULL )
116     fclose(rfp);
117   if( wfp != NULL )
118     fclose(wfp);
119   return EXIT_SUCCESS;
120 }
```

In diesem Listing werden mehrere typische Anwendungsfälle demonstriert, bei denen das zeilenweise Einlesen und Schreiben hilfreich sein kann. In den Zeilen **(05)** bis **(10)** finden Sie mit `killNL()` eine Funktion, mit der Sie das Newline-Zeichen aus einem String entfernen. Es wird bei `fgets()` von der Standardeingabe mit eingelesen (außer wenn der Dateiname bei den Beispielen `LINEBUF`-1 ist), ist aber bei den Dateinamen nicht erwünscht. Anstelle des Newline-Zeichens setzen Sie hiermit ggf. einfach das Stringende-Zeichen.

Mit der Funktion `countLineOut()` in den Zeilen **(16)** bis **(24)** können Sie die Zeilen einer Datei mit den Zeilennummern ausgeben lassen. Hierbei wird in der `while`-Schleife (Zeilen **(19)** bis **(23)**) so lange zeilenweise eingelesen und wieder ausgegeben, wie die Funktion `fgets()` ungleich `NULL` zurückgibt. In der `main()`-Funktion wurde zusätzlich noch die Option angeboten, diese Ausgabe der Funktion `countLineOut()` in eine Datei anstatt in die Standardausgabe zu schreiben. Hierbei müssen Sie lediglich die Standardausgabe in der Zeile **(87)** mittels `freopen()` umleiten.

In den Zeilen **(25)** bis **(40)** finden Sie mit der Funktion `search()` eine einfache Möglichkeit, innerhalb einer Datei nach Zeilen mit einer bestimmten Stringfolge zu suchen. Es wird ebenfalls Zeile für Zeile in einer `while`-Schleife durchlaufen und in jeder Zeile nach einer bestimmten Stringfolge mit der Funktion `strstr()` gesucht. Wurde eine Stringfolge gefunden, wird diese Zeile mit der Zeilennummer ausgegeben.

Mit der Funktion `copyFile()` sehen Sie in den Zeilen **(41)** bis **(51)** einen Klassiker. Mittels `fgets()` und `fputs()` können Sie ganz einfach zeilenweise den Inhalt von einem Lese-Stream in einen Schreib-Stream kopieren.

14.6.5 Lesen und Schreiben in ganzen Blöcken

Die Funktionen `fread()` und `fwrite()` sind speziellere Funktionen, die nicht mit einzelnen Zeichen oder Strings arbeiten, sondern in einzelnen Blöcke als Array.

Wenn Sie Strukturen speichern und wieder auslesen wollen, dann sind `fread()` und `fwrite()` dazu die perfekten Funktionen.

Die Syntax zum blockweisen Lesen mit `fread()` lautet:

```
#include <stdio.h>
size_t fread( void *buffer, size_t size, size_t n, FILE *fp );
```

Mit der Funktion `fread()` lesen Sie `n` Elemente der Größe `size` aus dem Stream `fp` aus und schreiben diese in das Array, auf das der Zeiger `buffer` zeigt. Wurden weniger als `n` Elemente eingelesen, kann es sein, dass vorher das Dateiende erreicht wurde oder ein Fehler auftrat.

Jetzt zur Syntax von `fwrite()`, dem Gegenstück von `fread()` zum Schreiben:

```
#include <stdio.h>
size_t fwrite( const void *buffer, size_t size,
               size_t n, FILE *fp );
```

Sie schreiben `n` Elemente mit der Größe von `size` aus dem durch `buffer` adressierten Speicherbereich in den Ausgabe-Stream `fp`. Hier ist der Rückgabewert die Anzahl der erfolgreich geschriebenen Elemente. Wurden weniger als `n` Elemente geschrieben, ist ein Fehler aufgetreten.

Nachfolgend sehen Sie ein Beispiel, wie Sie Datensätze von Strukturen speichern und wieder lesen können, ohne auf verkettete Listen oder Arrays von Strukturen zurückgreifen zu müssen, indem Sie einen neuen Datensatz direkt in eine Datei schreiben bzw. direkt daraus lesen können. Wie immer handelt es sich hier um ein für das Buch vereinfachtes Beispiel:

```
00  // kap014/listing005.c
01  #include <stdio.h>
02  #include <stdlib.h>
03  #include <string.h>
```

```
04 #define BUF 256
05 #define DATAFILE "data.dat"

06 void killNL( char *str ) {
07   size_t p = strlen(str);
08   if(str[p-1] == '\n') {
09     str[p-1] = '\0';
10   }
11 }

12 void dump_buffer(FILE *fp) {
13   int ch;
14   while( (ch = fgetc(fp)) != EOF && ch != '\n' )
15   /* Kein Anweisungsblock nötig */ ;
16 }

17 typedef struct _plz {
18   char ort[BUF];
19   unsigned int plz;
20 } Plz_t;

21 void newPLZ(void) {
22   Plz_t data;
23   printf("Ort          :  ");
24   if( fgets(data.ort, BUF, stdin) == NULL ) {
25     fprintf(stderr, "Fehler bei der Eingabe\n");
26     return;
27   }
28   killNL(data.ort);
29   printf("Postleitzahl :  ");
30   if( scanf("%u", &data.plz) != 1 ) {
31     fprintf(stderr, "Fehler bei der Eingabe\n");
32     return;
33   }
34   dump_buffer(stdin);
35   FILE *fp = fopen(DATAFILE, "a+b");
```

```
36    if( fp == NULL ) {
37      printf("Fehler beim Oeffnen: %s\n", DATAFILE);
38      exit(EXIT_FAILURE);
39    }
40    if(fwrite(&data, sizeof(data), 1, fp) != 1) {
41      fprintf(stderr,"Fehler beim Schreiben in %s\n",
          DATAFILE);
42      fclose(fp);
43      return;
44    }
45    fclose(fp);
46  }

47  void printPLZ(void) {
48    Plz_t data;
49    FILE *fp = fopen(DATAFILE, "r+b");
50    if( fp == NULL ) {
51      fprintf(stderr, "Fehler beim Oeffnen: %s\n", DATAFILE);
52      return;
53    }
54    while(fread(&data, sizeof(data), 1, fp) == 1 ) {
55      printf("\nOrt          : %s\n", data.ort);
56      printf("Postleitzahl   : %u\n\n", data.plz);
57    }
58    fclose(fp);
59  }

60  int main(void) {
61    int input = 0;
62    do {
63      printf("-1- Neuer Datensatz\n");
64      printf("-2- Datensaetze ausgeben\n");
65      printf("-3- Programm beenden\n\n");
66      printf("Ihre Auswahl : ");
67      if( scanf("%d", &input) != 1 ) {
68        fprintf(stderr,"Fehler bei der Eingabe\n");
```

```
69        input = 0;
70      }
71      dump_buffer(stdin);
72      switch( input ) {
73        case 1 : newPLZ( );   break;
74        case 2 : printPLZ( ); break;
75      }
76    }while(input!=3);
77    return EXIT_SUCCESS;
78  }
```

Zur Vereinfachung wurde in den Zeilen **(17)** bis **(20)** ein Strukturtyp deklariert, der Postleitzahlen mit deren Ort und Nummer speichert. Das Programm selbst besteht aus zwei Funktionen, zum einen aus einer Schreibfunktion `newPLZ()`, die in den Zeilen **(21)** bis **(46)** definiert wurde. In dieser Funktion werden jeweils die Daten für die Struktur in einer Strukturvariablen `Plz_t` eingelesen (Zeilen **(23)** bis **(34)**). Anschließend wird eine Datei in Zeile **(35)** im Modus `"a+b"` geöffnet, in welcher die Daten der Strukturvariablen binär geschrieben und immer an das Ende angehängt werden sollen. Existiert diese Datei noch nicht, wird sie angelegt. In Zeile **(40)** wird die Strukturvariable als Ganzes in den Stream geschrieben. Am Ende wird der Stream in Zeile **(45)** wieder ordnungsgemäß geschlossen.

Die zweite Funktion `printPLZ()` liest diese Daten der Datei wieder aus. Zunächst wird die Datei in Zeile **(49)** zum binären Lesen geöffnet und in den Zeilen **(54)** bis **(57)** Datensatz für Datensatz gelesen. Solange `fread()` in der Zeile **(54)** 1 für einen erfolgreich gelesenen Datensatz zurückgibt, wurde einen kompletter Datensatz `Plz_t` gefunden und ausgegeben. Am Ende wird der Stream in Zeile **(58)** mit `fclose()` wieder geschlossen.

14.7 Funktionen zur formatierten Ein-/Ausgabe

Zum formatierten Einlesen und Schreiben von Daten stehen Ihnen verschiedene `scanf`- und `printf`-Funktionen in der Headerdatei `<stdio.h>` zur Verfügung. Beide Funktionsfamilien bieten sogenannte Umwandlungs-

vorgaben des Formatstrings an, mit denen Sie das Datenformat steuern können.

14.7.1 Funktionen zur formatierten Ausgabe

Die einfache `printf`-Funktion haben Sie ja schon des Öfteren in diesem Buch verwendet. Neben dieser Version gibt es noch einige andere Versionen, die alle auf recht ähnliche Weise den Formatstring verwenden können. Folgende `printf`-Versionen stehen ihnen zur Verfügung:

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

Die grundlegende `printf`-Funktion schreibt den String `format` auf die Standardausgabe `stdout`. Die Version mit Stream hat folgende Syntax:

```
int fprintf( FILE* restrict fp,
             const char* restrict format, ... );
```

Damit schreiben Sie den String `format` auf den Ausgabe-Stream `fp`. `printf(format)` entspricht `fprintf (stdout, format)`. Es ist sehr hilfreich, etwas formatiert in eine Datei zu schreiben. Des Weiteren gibt es auch zwei String-Versionen. Die Syntax dazu:

```
int sprintf( char * restrict buf,
             const char * restrict format, ... );
int snprintf( char * restrict buf,
              size_t n,
              const char * restrict format, ... );
```

Hiermit schreiben Sie `format` in einen String mit der Anfangsadresse von `buf`. Die Version `snprintf()` begrenzt die Zeichen, die nach `buf` geschrieben werden, zudem noch auf `n` Bytes. Das stellt die sicherere Alternative gegenüber `sprintf()` dar. Des Weiteren gibt es noch Versionen all der eben erwähnten `printf`-Varianten wie `vprintf`, `vfprintf`, `vsprintf` und `vsnprintf` mit dem Präfix *v* für Parameter aus variabler Parameterliste, worauf in diesem Buch allerdings nicht näher eingegangen wird.

Drei Punkte von printf

Die drei Punkte in den `printf`-Anweisungen (auch als Ellipse bekannt) stehen für weitere optionale Argumente, die hier neben dem festen Argument `format` verwendet werden können.

Umwandlungsvorgaben für die printf-Familie

Bei den Datentypen haben Sie die Umwandlungsvorgaben zu den einzelnen Typen näher kennengelernt, die mit dem Zeichen `%` beginnen und mit einem Buchstaben, dem Konvertierungsspezifizierer, enden. Sie beziehen sich dann auf das nachfolgende Argument. `%d` steht beispielsweise für den Spezifizierer des Integerwertes (`int`). Folglich wird in der Argumentenliste auch ein solcher Typ zur Konvertierung im Formatstring passen. Hier ein Beispiel:

```
int val = 12345;
char str[] = "Hallo Welt";
printf("Der Wert ist %d, und der String lautet %s", val, str);
```

Neben den einfachen Umwandlungszeichen für den entsprechenden Typ zur Konvertierung im Formatstring gibt es noch einige Möglichkeiten mehr, den Formatstring zu formatieren. Folgende Syntax soll hier für die anschließende Beschreibung verwendet werden:

```
%[F][W][G][L]U
```

Die einzelnen Teile in den eckigen Klammern können optional zu `U` verwendet werden. Die Bedeutung der einzelnen Buchstaben ist:

- `F` = Flags
- `W` = Weite, Feldbreite
- `G` = Genauigkeit
- `L` = Längenangabe
- `U` = Umwandlungszeichen, Konvertierungsspezifizierer

Regel für eine Konvertierungsspezifikation

Die Syntax einer Konvertierungsspezifikation, die mit dem Zeichen % beginnt, endet immer mit einem Konvertierungsspezifizierer, also dem Buchstaben (Umwandlungszeichen) für den entsprechenden Typ.

Weite, Feldbreite

Die wohl am häufigsten verwendete Konvertierungsspezifikation dürfte die Feldbreite sein. Sie kann bei jedem Umwandlungstyp verwendet werden. Damit können Sie festlegen, wie viele Zeichen zur Ausgabe mindestens verwendet werden dürfen. Zur Angabe der Feldbreite wird entweder eine Ganzzahl oder ein Stern (*) verwendet. Mit dem Sternchen können Sie die Feldbreite variabel halten und müssen dafür ein zusätzliches Argument in der Argumentenliste vom Typ `int` angeben, dessen Wert als Feldbreite verwendet wird.

Standardmäßig erfolgt die Ausgabe rechtsbündig, wobei eine zu große Feldbreite mit Leerzeichen auf der linken Seite aufgefüllt wird. Für eine linksbündige Ausrichtung setzen Sie vor der Feldbreite das Minuszeichen (ein Flag). Wenn Sie eine zu kleine Feldbreite angeben, bewirkt dies bei der Ausgabe nicht, dass die Zahlen beschnitten bzw. weniger Zeichen ausgegeben werden. Sie werden dennoch komplett ausgegeben, d. h. die Angabe wird ignoriert. Hierzu ein einfaches Listing, mit dem die Wirkung der Feldbreite demonstriert wird:

```
00  // kap014/listing006.c
01  #include <stdio.h>
02  #include <stdlib.h>

03  int main (void) {
04    char text[] = "Tiefstand";
05    int breite = 20;
06    printf("|01234567890123456789|\n");
07    printf("|%s|\n", text);
```

```
08    printf("|%20s|\n", text);
09    printf("|%-20s|\n", text);
10    printf("|%20s|\n", text+4); // Textanfang um 4 Zeichen nach vorne
11    printf("|%-20s|\n", text+4);
12    printf("|%*s|\n", breite, text);
13    return EXIT_SUCCESS;
14  }
```

Das Programm bei der Ausführung:

```
|01234567890123456789|
|Tiefstand|
|           Tiefstand|
|Tiefstand           |
|               stand|
|stand               |
|           Tiefstand|
```

Flags

Die Flags stehen unmittelbar nach dem %-Zeichen. Wenn es sinnvoll ist, können mehrere Flags gleichzeitig verwendet werden. In der Tabelle 14.3 finden Sie die Flags und deren Bedeutung aufgelistet.

Flag	Beschreibung
-	linksbündig justieren
+	Ausgabe des Vorzeichens. + wird bei positiven Werten vorangestellt, – bei negativen Werten.
Leerzeichen	Ist ein Argument kein Vorzeichen, wird ein Leerzeichen ausgegeben.
0	Bei numerischer Ausgabe wird mit Nullen bis zur angegebenen Feldbreite gefüllt.

Tabelle 14.3 Flags für die Formatierungsanweisung

Flag	Beschreibung
#	Die Wirkung des Zeichens # hängt vom verwendeten Format ab. Wenn Sie zum Beispiel einen Wert über %x als Hexadezimal ausgeben, wird bei %#x dem Wert ein 0x vorangestellt. Bei e, E oder f wird der Wert mit einem Dezimalpunkt ausgegeben, auch wenn keine Nachkommastelle existiert.

Tabelle 14.3 Flags für die Formatierungsanweisung (Forts.)

Zum besseren Verständnis der einzelnen Flags soll auch hierzu wieder ein einfaches Listing mit deren Anwendung gezeigt werden:

```
00  // kap014/listing007.c
01  #include <stdio.h>
02  #include <stdlib.h>

03  int main (void) {
04    int val = 123456789;
05    printf("|012345678901|\n");
06    printf("|%d|\n", val);
07    printf("|%+12d|\n", val);
08    printf("|%-12d|\n", val);
09    printf("|%012d|\n", val);
10    printf("|%-#12o|\n", val);
11    printf("|%#x|\n", val);
12    printf("|%#012X|\n", val);
13    return EXIT_SUCCESS;
14  }
```

Das Programm bei der Ausführung:

```
|012345678901|
|123456789|
|  +123456789|
```

```
|123456789    |
|000123456789|
|0726746425   |
|0x75bcd15|
|0X00075BCD15|
```

Genauigkeit

Geben Sie einen Punkt an, gefolgt von einer Ganzzahl, können Sie die Genauigkeitsangabe der Nachkommastelle von Gleitpunktzahlen oder einem String beschneiden. Natürlich wird hierbei nicht der Wert selbst verändert, sondern die Angaben beziehen sich immer nur auf die formatierte Ausgabe. Bei Ganzzahlen und Strings hat eine kleinere Genauigkeitsangabe keine Wirkung auf die Ausgabe. Hierzu wieder ein kleines Beispiel, das Ihnen die Genauigkeitsangabe etwas näherbringt:

```
00  // kap014/listing008.c
01  #include <stdio.h>
02  #include <stdlib.h>

03  int main (void) {
04    float fval = 3.14132;
05    char text[] = "Tiefstand";
06    printf("|%08.2f|\n", fval);
07    printf("|%-8.0f|\n", fval);
08    printf("|%8.4s|\n", text);
09    printf("|%-8.4s|\n", text);
10    return EXIT_SUCCESS;
11  }
```

Das Programm bei der Ausführung:

```
|00003.14|
|3       |
|    Tief|
|Tief    |
```

Umwandlungszeichen

Zwar haben Sie die Umwandlungszeichen bereits mit den Datentypen näher kennengelernt, aber trotzdem sollen sie hier zur besseren Übersicht nochmals alle aufgelistet werden.

Zeichen	Typ	Bedeutung
`%c`	`char`, `int`	Einzelnes Zeichen. Für die numerische Ausgabe von `signed char` werden `%hhi` und `unsigned char` `%hhu` verwendet.
`%lc`	`wchar_t`	Einzelnes Breitzeichen bei entsprechender w-Funktion
`%s`	Zeiger auf `char`	Ein String (char-Array)
`%p`		Ausgabe des Zeigerwertes (Adresse)
`%d`, `%i`	`int`	Vorzeichenbehaftete dezimale Ganzzahl
`%u`	`unsigned int`	Vorzeichenlose Ganzzahl
`%o`	`unsigned int`	Oktale Darstellung einer vorzeichenlosen Ganzzahl
`%x`	`unsigned int`	Hexadezimale Darstellung einer vorzeichenlosen Ganzzahl mit den Buchstaben a, b, c, d, e, f
`%X`	`unsigned int`	Hexadezimale Darstellung einer vorzeichenlosen Ganzzahl mit den Buchstaben A, B, C, D, E, F

Tabelle 14.4 Umwandlungszeichen für die einzelnen Typen

Zeichen	Typ	Bedeutung
%lld, %lli	long long	Vorzeichenbehaftete Ganzzahl
%llu	unsigned long long	Vorzeichenlose Ganzzahl
%llx, %llX	unsigned long long	Hexadezimale Darstellung einer vorzeichenlosen Ganzzahl mit den Buchstaben a, b, c, d, e, f bzw. A, B, C, D, E, F
%f, %lf (für scanf)	double	Dezimale Gleitpunktzahl in Form von dd.dddd
%e, %E	double	Exponentiale Darstellung der Gleitpunktzahl in Form von d.dde+-dd bzw. d.ddE+-dd. Der Exponent enthält mindestens zwei Ziffern.
%g, %G	double	Gleitpunkt- oder Exponentialdarstellung, abhängig davon, was kürzer ist
%a, %A	double	Exponentialdarstellung der Gleitpunktzahl in hexadezimaler Form
%Lf	long double	Gleitpunktdarstellung, die für Gleitpunktzahlen mit einer erweiterten Genauigkeit verwendet wird

Tabelle 14.4 Umwandlungszeichen für die einzelnen Typen (Forts.)

Neben den Umwandlungsvorgaben gibt es sogenannte Argumenttyp-Modifikationen. Folgende Typen gibt es, und folgende Auswirkungen haben sie auf die damit verwendeten Umwandlungszeichen:

Modifikation	Auswirkung
`h`	Die Umwandlungszeichen d, i, o, u, x, X werden als `short int`-bzw. `unsigned short int`-Wert behandelt (beispielsweise `%hd`).
`l`	Die Umwandlungszeichen d, i, o, u, x, X werden als `long int`- bzw. `unsigned long int`-Wert behandelt. Wird hingegen e, f oder g verwendet, werden die Umwandlungszeichen als `double`-Wert behandelt (beispielsweise `%lf`).
`ll`	Die Umwandlungszeichen d, i, o, u, x, X werden als `long long int`- bzw. `unsigned long long int`-Wert behandelt (seit C99).
`L`	Die Umwandlungszeichen e, E, f, g, G werden als `long double`-Wert behandelt. Die Umwandlungszeichen d, i, o, u, x, X hingegen werden als `long long`-Wert behandelt (beispielsweise `%Ld`).
`hh`	Wie h, nur dass die Umwandlungszeichen d, i, o, u, x, X als `signed char`- bzw. `unsigned char`-Wert behandelt werden (beispielsweise `%hhd`) (seit C99).
`j`	Die Umwandlungszeichen d, i, o, u, x, X werden als `intmax_t`- bzw. `uintmax_t`-Wert behandelt (seit C99).
`t`	Die Umwandlungszeichen d, i, o, u, x, X werden als `ptrdiff_t`-Wert behandelt (seit C99).
`z`	Die Umwandlungszeichen d, i, o, u, x, X werden als `size_t`-Wert behandelt (seit C99).

Tabelle 14.5 Argumenttyp-Modifikationen

14.7.2 Funktionen zur formatierten Eingabe

Wie bei der printf-Familie für die Ausgabe gibt es bei der scanf-Familie für die Eingabe mehrere Funktionen. Alle Funktionen können Sie auf eine recht ähnlich Weise verwenden, wie Sie es von scanf bereits kennen. Nur die Angaben der Quelle und der Argumentübergabe sind dabei anders. Hier wieder ein Überblick über die einzelnen scanf-Funktionen:

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

Damit lesen Sie von der Standardeingabe stdin formatiert ein. Natürlich gibt es wieder ein Stream-orientiertes Gegenstück. Die Syntax hierzu lautet:

```
int fscanf( FILE* restrict fp,
            const char * restrict format, ... );
```

Die Funktion entspricht der einfachen scanf-Funktion, nur wird hierbei vom Eingabe-Stream fp eingelesen. fscanf(stdin, format) hat somit denselben Effekt wie die einfache scanf-Funktion. Auch eine String-orientierte Version ist vorhanden. Die Syntax hierzu:

```
int sscanf( const char * restrict src,
            const char * restrict format, ... );
```

Damit erfolgt die Eingabe von einem String. Auch hier existieren noch die Versionen vscanf, vfscanf und vsscanf, bei denen die Eingabe aus stdin, einer Datei bzw. aus einem String kommt und für Parameter aus variabler Parameterliste verwendet wird. Auch hierauf wird nicht mehr näher eingegangen.

Umwandlungsvorgaben für die scanf-Familie

Die gängigen Umwandlungszeichen zu scanf finden Sie in der Tabelle 14.4 im vorigen Abschnitt zur printf-Familie. Zwar sind die Umwandlungszeichen von printf und scanf recht ähnlich, aber nicht (!) identisch. Sie müssen beispielsweise darauf achten, dass Sie für ein double die Umwandlungsvorgabe %lf bei scanf verwenden. Bei printf ist hier hingegen %f erlaubt.

Daher wird in diesem Abschnitt darauf verzichtet, nochmals auf die einzelnen Umwandlungszeichen einzugehen. Sie können den Formatstring mit unterschiedlichen `scanf`-Funktionen formatieren. Es gilt folgende allgemeine Syntax:

```
%[W][L][S]U
```

Die einzelnen Teile in den eckigen Klammern können optional zu `U` verwendet werden. Die Bedeutung der einzelnen Buchstaben ist:

- `W` = Weite, Feldbreite
- `L` = Längenangabe
- `S` = Suchmengenkonvertierung
- `U` = Umwandlungszeichen, Konvertierungsspezifizierer

Die Konvertierungsspezifikationen von `W` (für Weite, Feldbreite), `L` (für Längenangabe) und `U` (für Umwandlungszeichen) haben Sie bereits ausführlich im Unterabschnitt »Umwandlungsvorgaben für die printf-Familie« in Abschnitt 14.7.1, »Funktionen zur formatierten Ausgabe«, kennengelernt. Die Bedeutung ist meistens recht ähnlich, nur bezieht sich diese hier auf die Eingabe statt auf die Ausgabe.

Suchmengenkonvertierung

Zusätzlich bieten die `scanf`-Funktionen beim Einlesen von Strings sogenannte Suchmengenkonvertierungen an. Sie können anstelle des Umwandlungszeichens `s` (für Strings, `char`-Arrays) verwendet werden.

Suche	Es wird eingelesen, ...
`%[bezeichner]`	... bis ein Zeichen vorkommt, das nicht in der Liste `bezeichner` steht.
`%[^bezeichner]`	... bis ein Zeichen vorkommt, das in der Liste `bezeichner` steht.

Tabelle 14.6 Suchmengenkonvertierung für die scanf-Funktionen

Ein kurzer Codeausschnitt hierzu:

```
01 char str[255], str2[255];
02 printf("Nur Zahlen eingeben    : ");
03 if( scanf("%254[0-9]", str) != 1 ) {
04   fprintf(stderr, "Fehler bei der Eingabe\n");
05   return EXIT_FAILURE;
06 }
07 dump_buffer(stdin);
08 printf("Keine Zahlen eingeben  : ");
09 if( scanf("%254[^0-9]", str2) != 1 ) {
10   fprintf(stderr, "Fehler bei der Eingabe\n");
11   return EXIT_FAILURE;
12 }
```

Wenn Sie in Zeile **(03)** etwas eingeben, wird so lange eingelesen, bis das erste Zeichen nicht 0 bis 9 ist. Verwenden Sie dasselbe mit dem Caret-Zeichen (^), wird so lange eingelesen, bis ein Zeichen 0 bis 9 ist (siehe Zeile **(09)**). Wenn Sie das Beispiel ausprobieren, werden Sie feststellen, dass `scanf` in der Zeile **(09)** nicht wie üblich beim Auftreten eines Whitespace-Zeichens wie Leerzeichen, Tabs oder Zeilenumbrüchen abgebrochen wird. Zusätzlich wurde hier auch die Feldbreite mit 254 in den Zeilen **(03)** und **(09)** verwendet, damit kein Pufferüberlauf stattfindet. Dabei muss allerdings angemerkt werden, dass sich die weiteren Zeichen darüber hinaus im Eingabepuffer des Programms befinden, wenn mehr als 254 Zeichen eingegeben wurden. Dies sollten Sie wissen, wenn Sie beispielsweise vorhaben sollten, gleich darauf ein weiteres `scanf` aufzurufen.

Weitere mögliche Beispiele zur Suchmengenkonvertierung:

```
%[A-Z]    // Großbuchstaben von A bis Z
%[a-fm-z] // Kleinbuchstaben a bis f und m bis z
%[a-fA-F] // Groß- und Kleinbuchstaben a-f A-F (Hexadezimal)
```

Bei der Verwendung eines Bindestrichs müssen Sie darauf achten, dass die Zeichen lexikalisch nach der ASCII-Tabelle verwendet werden. Es kann also nicht `C-A`, sondern es muss immer `A-C` verwendet werden. Natürlich

setzt die Verwendung von A-Z auch wieder voraus, dass die ASCII-Code-Tabelle für die Kodierung von Zeichen verwendet wird, was der Standard ja nicht vorgibt.

14.8 Wahlfreier Dateizugriff

Wenn Sie eine Datei öffnen, verweist ein Indikator für die Dateiposition (der Schreib-/Lesezeiger) auf den Anfang der Datei, genauer gesagt: auf das erste Zeichen mit der Position 0. Öffnen Sie eine Datei im Anhängemodus (a bzw. a+), verweist der Schreib-/Lesezeiger auf das Ende der Datei. Mit jeder Lese- oder Schreiboperation erhöht sich auch der Schreib-/Lesezeiger um die Anzahl der übertragenen Zeichen. Möchten Sie diesen sequenziellen Arbeitsfluss von Dateien ändern, müssen Sie Funktionen für einen wahlfreien Dateizugriff verwenden. Hierfür stehen die Funktionen fseek(), rewind() und fsetpos() zur Verfügung.

14.8.1 Dateiposition ermitteln

Müssen Sie eine aktuelle Dateiposition im Programm abfragen, und wollen Sie beispielsweise erst später wieder auf diese Position zugreifen, bietet C in der Headerdatei <stdio.h> zwei verschiedene Funktionen an. Zunächst die Funktion ftell():

```
long int ftell( FILE* fp );
```

Diese Funktion liefert die aktuelle Schreib-/Lese-Position des Streams fp zurück. Erhalten Sie einen Fehler zurück, dann ist der Wert –1L. Ähnlich funktioniert auch die Funktion fgetpos():

```
int fgetpos( FILE* restrict fp, fpos_t* restrict pos );
```

Bei dieser Funktion wird die aktuelle Schreib-/Lese-Position des Streams fp an das mit pos referenzierte Speicherobjekt vom Typ fpos_t (meistens ein typedef auf long oder long long) geschrieben. Die Funktion gibt 0 zurück, wenn alles in Ordnung war, ansonsten wird ein Wert ungleich 0 zurückgegeben.

14.8.2 Dateiposition ändern

Die Position von Schreib-/Lesezeigern ändern Sie durch drei in der Headerdatei definierte Funktionen. Zunächst die Syntax der Funktion `fseek()`:

```
#include <stdio.h>
int fseek( FILE* fp, long offset, int origin );
```

Damit wird der Schreib-/Lesezeiger vom Stream `fp` durch die Angaben von `offset` relativ vom Bezugspunkt `origin` versetzt. Für `origin` sind folgende drei Konstanten definiert:

Makro	Bezugspunkt ab ...
SEEK_SET	... dem Anfang der Datei
SEEK_CUR	... der aktuellen Position
SEEK_END	... dem Ende der Datei

Tabelle 14.7 Konstanten für den Bezugspunkt »origin« in der Funktion »fseek()«

Trat bei der Funktion `fseek()` kein Fehler auf, wird 0 zurückgegeben und ggf. auch das `EOF`-Flag gelöscht. Im Fehlerfall ist der Rückgabewert der Funktion ungleich 0.

Die zweite Funktion zum Ändern des Schreib-/Lesezeigers und so etwas wie das Gegenstück zur Funktion `fgetpos()` ist die Funktion `fsetpos()`. Hier die Syntax dazu:

```
#include <stdio.h>
int fsetpos( FILE* fp, const fpos_t* pos );
```

Hiermit setzen Sie den Schreib-/Lesezeiger des Streams `fp` auf den Wert, der durch `pos` referenziert wird. In der Regel sollte diese Position ein Wert von der zuvor aufgerufenen Funktion `fgetpos()` sein. Wurde die Funktion erfolgreich ausgeführt, wird 0 zurückgegeben und ebenfalls das Flag des *end-of-file* und *error indicator* gelöscht. Bei einem Fehler wird ein Wert ungleich 0 zurückgegeben.

Mit der Funktion `rewind()` gibt es noch eine dritte Funktion zum Verändern der Dateiposition. Wie Sie am Namen schon ablesen können, setzt diese Funktion den Schreib-/Lesezeiger auf den Anfang der Datei. Hier die Syntax dazu:

```
#include <stdio.h>
void rewind( FILE* fp );
```

Nachdem der Schreib-/Lesezeiger des Streams `fp` auf den Anfang gesetzt wurde, wird auch das Flag des *end-of-file* und *error indicator* gelöscht. Ein Aufruf von `rewind(fp)` entspricht folgendem Aufruf:

```
(void)fseek( fp, 0L, SEEK_SET );
```

Hierzu ein einfaches Beispiel, das Ihnen den wahlfreien Dateizugriff in der Praxis demonstriert:

```
00  // kap014/listing009.c
01  #include <stdio.h>
02  #include <stdlib.h>
03  #define FILENAME "listing009.c" // Anpassen

04  void dump_buffer(FILE *fp) {
05    int ch;
06    while( (ch = fgetc(fp)) != EOF && ch != '\n' )
07    /* Kein Anweisungsblock nötig */ ;
08  }

09  long fileSize( FILE *fp ) {
10    if( fseek( fp, 0L, SEEK_END ) != 0 ) {
11      fprintf(stderr, "Fehler bei fseek\n");
12      return -1;
13    }
14    return ftell(fp);
15  }

16  int main (void) {
```

```
17    long endPos = 0, aktPos = 0, neuPos = 0;
18    FILE *rfp = fopen(FILENAME, "r");
19    if( rfp != NULL ) {
20      endPos = fileSize( rfp );
21      if(endPos == -1 ) {
22        return EXIT_FAILURE;
23      }
24    }
25    else {
26      fprintf(stderr, "Fehler beim Oeffnen\n");
27      return EXIT_FAILURE;
28    }
29    rewind( rfp ); // Wieder zum Anfang zurück
30    printf("Lesezeiger nach vorne setzen. Um wie viel: ");
31    if (scanf("%ld", &neuPos) != 1 ) {
32      fprintf(stderr, "Fehler bei der Eingabe\n");
33      return EXIT_SUCCESS;
34    }
35    dump_buffer(stdin);
36    if( neuPos > endPos ) {
37      fprintf(stderr,
          "Fehler: Datei ist nur %ld Byte groß\n",endPos);
38    }
39    if( fseek( rfp, neuPos, SEEK_CUR ) != 0 ) {
40      fprintf(stderr, "Fehler bei fseek\n");
41      return EXIT_FAILURE;
42    }
43    int c;
44    while( (c=fgetc(rfp)) != EOF ) {
45      fputc(c, stdout);
46    }
47    fclose(rfp);
48    return EXIT_SUCCESS;
49  }
```

Nachdem eine Datei in Zeile **(18)** geöffnet wurde, holen Sie die Dateigröße in Zeile **(20)** mit der selbst geschriebenen Funktion `fileSize()`. Die Funktion wurde in den Zeilen **(09)** bis **(15)** definiert. In der Funktion wird zunächst in Zeile **(10)** der Stream mit `fseek()` auf das Dateiende gesetzt. Diese Position wird in Zeile **(14)** mit der Funktion `ftell()` an den Aufrufer zurückgegeben. In der Variablen `endPos` in Zeile **(20)** steht dann quasi die Dateigröße in Bytes. In Zeile **(29)** setzen Sie den Lesezeiger des Streams mit `rewind()` wieder auf den Anfang der Datei.

In den Zeilen **(30)** und **(31)** werden Sie gefragt, wie viele Bytes Sie dem Anfang des Lesezeigers vorsetzen wollen. Den eingegebenen Wert überprüfen Sie zunächst in Zeile **(36)** mit der zuvor ermittelten Dateigröße. Damit verschieben Sie den Lesezeiger nicht über die Dateigröße hinaus, denn das würde zu einem Fehler führen. In Zeile **(39)** wird der Lesezeiger des Streams dann von der aktuellen Position, dem Dateianfang, auf die neue Position gesetzt und in den Zeilen **(44)** bis **(46)** Zeichen für Zeichen eingelesen und ausgegeben.

14.9 Sicherere Funktionen mit C11

In Abschnitt 9.3.5, »Sicherere Funktionen zum Schutz vor Speicherüberschreitungen«, haben Sie ja bereits erfahren, dass es seit C11 als Erweiterung (!) ein sogenanntes *Bound-Checking-Interface* gibt. Hierbei stehen Ihnen alternative Funktionen zu bereits vorhandenen »unsicheren« Funktionen der Standardbibliothek mit der Endung _s zur Verfügung (beispielsweise `scanf_s` für `scanf`). Werfen Sie ein Blick in Annex K des C11-Standards (*http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1570.pdf*): Dort wird das Thema umfassender behandelt. Diese neuen Funktionen prüfen beispielsweise die Puffergröße und geben Fehlermeldungen aus, wenn der Puffer nicht groß genug ist. Auch Ergebnisstrings werden immer mit den Stringende-Zeichen abgeschlossen, oder die Rückgabewerte sind einheitlich vom Typ `errno_t` und geben Informationen über den erfolgreichen oder nicht erfolgreichen Aufruf der Operation zurück.

Da dieses Bound-Checking-Interface eine optionale C11-Erweiterung ist, müssen Sie prüfen, ob das Makro `__STDC_LIB_EXT1__` gleich 1 ist, ob die Erweiterung vorhanden ist und verwendet werden kann. Ist die Erweite-

rung vorhanden, müssen Sie nur die entsprechenden und bekannten Ein- und Ausgabe-Funktionen durch ihr jeweiliges Gegenstück mit der Endung `_s` ersetzen.

14.10 Datei löschen oder umbenennen

Möchten Sie eine Datei löschen oder umbenennen, stehen Ihnen in der Headerdatei `<stdio.h>` zwei Funktionen zur Verfügung, die keinen Stream benötigen. Eine Datei löschen können Sie mit der Funktion:

```
int remove( const char *pathname );
```

Bei Erfolg gibt die Funktion 0 zurück. Bei einem Fehler, wenn die Datei nicht gelöscht werden konnte, gibt sie einen Wert ungleich 0 zurück. Neben der richtigen Pfadangabe sind zum Löschen natürlich auch die erforderlichen Zugriffsrechte nötig.

Ähnlich einfach ist die Funktion zum Umbenennen von Dateien aufgebaut. Die Syntax hierzu:

```
int rename( const char *oldname, const char *newname );
```

Wenn alles glatt verlief, wird der Name `oldname` durch `newname` ersetzt, und die Funktion gibt 0 zurück. Im Fehlerfall wird ungleich 0 zurückgegeben.

14.11 Pufferung

Eine kurze Erklärung noch zur Pufferung: Die Standardeinstellung ist bei ANSI-C-Compilern die Vollpufferung. Das ist sinnvoller und schneller, als einzelne Zeichen zu lesen oder zu schreiben, denn es finden weniger Lese- und Schreiboperationen etwa auf der Festplatte oder auf dem Arbeitsspeicher statt. Die Puffergröße ist abhängig vom Compiler, liegt aber meistens bei 512 und 4.096 Bytes. Die Größe ist in der Headerdatei `<stdio.h>` mit der Konstante `BUFSIZ` angegeben.

Das bedeutet allerdings nicht, dass Sie von der Vollpufferung abhängig sind. Wenn Sie eine andere Pufferung für einen Stream verwenden möch-

ten, können Sie dies mit den Funktionen `setbuf()` oder `setvbuf()` nach dem Öffnen eines Streams mittels `fopen()` ändern. Folgende Pufferungsarten können Sie verwenden:

- Vollpufferung: Das ist die Standardeinstellung. Die Zeichen im Puffer werden erst übertragen, wenn der Puffer voll ist. Die Ausgabe von Daten, die noch im Puffer liegen, kann aber auch mit der Funktion `fflush()` erzwungen werden. Der Puffer wird nach dem normalen Schließen eines Streams und nach dem normalen Beenden eines Programms automatisch geleert.
- Zeilenpufferung: Hier werden die Daten im Puffer erst übertragen, wenn ein Newline-Zeichen vorhanden oder wenn der Puffer voll ist.
- Ungepuffert: Die Zeichen werden unmittelbar aus dem Stream übertragen.

fflush nur für Ausgabe-Streams

Die Funktion `fflush()` ist hilfreich für Ausgabe-Streams und sorgt dafür, dass alle noch im Puffer befindlichen und nicht geschriebenen Daten auf die Festplatte geschrieben werden. Nicht verwenden dürfen Sie die Funktion jedoch auf einen Eingabe-Stream, weil hier das weitere Verhalten laut Standard undefiniert ist. Ein `fflush(stdin)` mag zwar bei einigen Compilern funktionieren, um den Eingabepuffer der Tastatur zu »leeren«, aber es bleibt trotzdem dabei, dass die Verwendung dieser Funktion auf einen Eingabe-Stream per Standard nicht erlaubt ist.

14.12 Kontrollfragen und Aufgaben

1. Was sind Streams?
2. Welche Funktionen kennen Sie, um eine Datei zu öffnen?
3. Im folgenden Codeausschnitt soll die Datei *datei.txt* zeichenweise in die Datei *kopie.txt* kopiert werden. Allerdings sind nach diesem Vorgang beide Dateien leer. Was wurde falsch gemacht?

```
01  #define FILENAME1 "datei.txt"
02  #define FILENAME2 "kopie.txt"
    ...
03    FILE *fpr, *fpw;
04    int c;
05    fpr = fopen( FILENAME1, "w" );
06    fpw = fopen( FILENAME2, "a+" );
      ...
07    while ( (c=fgetc(fpr)) != EOF ) {
08      fputc(c, fpw );
09    }
      ...
```

4. Im folgenden Beispiel soll in einer Schleife eine Art Berechnung simuliert werden. Hierbei wird keine echte Berechnung durchgeführt. In Zeile **(10)** basteln Sie den Dateinamen mithilfe der Funktion `sprintf()` à la *Berechnung1.txt*, *Berechnung2.txt* usw. In den Zeilen **(11)** und **(12)** tun Sie einfach so, als fände eine Berechnung statt. In Zeile **(15)** erstellen Sie eine neue Datei für die Berechnung mit dem zuvor erzeugten Dateinamen von Zeile **(10)**, und schreiben Sie das Ergebnis der Berechnung in Zeile **(16)** in die Datei, in diesem Fall nur einen formatierten String mit der Nummer der Berechnung. Konnte keine Datei erzeugt werden, wird stattdessen das Ergebnis der Berechnung in Zeile **(19)** formatiert und auf die Standardfehlerausgabe, also den Bildschirm, ausgegeben. Auf einigen Systemen werden alle Berechnungen in eine Datei von *Berechnung1.txt* bis *Berechnung999.txt* geschrieben. Bei anderen Systemen werden ab einer gewissen Anzahl von Berechnungen nur noch die Ergebnisse auf die Standardfehlerausgabe gemacht, und es wird keine neue Datei mehr angelegt. Warum?

```
00  // kap014/aufgabe001.c
01  #include <stdio.h>
02  #include <stdlib.h>

03  int main(void) {
04    int i = 0;
05    char filename[255];
06    FILE *fp[1000];
```

```
07    printf("Programmstart\n");
08    while( i < 1000 ) {
09      // Dateinamen basteln mit fortlaufender Nummer
10      sprintf( filename, "Berechnung%d.txt", i);
11      // Umfangreiche Berechnung hier
12      // ...
13      // Ergebnis in eine Datei
14      fp[i] = fopen( filename, "w" );
15      if( fp[i] != NULL ) {
16        fprintf(fp[i], "Ergebnis der Berechnung Nr.%d", i );
17      }
18      else {
19        fprintf(stderr, "Ergebnis der Berechnung Nr.%d", i);
20      }
21      ++i;
22    }
23    return EXIT_SUCCESS;
24  }
```

5. Erstellen Sie ein Programm, das eine Datei mit folgendem Inhalt ausliest:

   ```
   Montag;2345;2341
   Dienstag;3245;1234
   Mittwoch;3342;2341
   Donnerstag;2313;2341
   Freitag;2134;6298
   ```

   Die Semikola stellen die Trennzeichen zwischen den einzelnen Werten dar. Folgende Bedeutung fällt den einzelnen Werten zu:

   ```
   Tag;Einnahmen;Ausgaben
   ```

   Berechnen Sie zusätzlich alle Einnahmen und Ausgaben am Ende der Woche, und erstellen Sie eine Gesamtbilanz, wie viel in der Woche eingenommen bzw. ausgegeben wurde. **Tipp:** Verwenden Sie `fscanf()` zum formatierten Einlesen.

Anhang A

Übersichtstabellen wichtiger Sprachelemente

A.1 Operator-Priorität (Operator Precedence)

In der folgenden Tabelle werden die Operatoren von C und ihre Assoziativität (die Bindung der Operanden) in absteigender Reihenfolge aufgelistet. Operatoren derselben Prioritätsklasse haben dieselbe Rangstufe.

Rang	Operator	Beschreibung	Richtung

1	`(Postfix)++`
	`(Postfix)--`
	`()`
	`[]`
	`.`
	`->`
	`(Typ){list}`
		Postfix-Inkrement
		Postfix-Dekrement
		Funktionsaufruf
		Indizierung
		Elementzugriff
		Elementzugriff
		Compound literal$_{(C99)}$
			Links nach rechts

2	`++(Präfix)`
	`--(Präfix)`
	`+`
	`-`
	`!`
	`~`
	`&`
	`*`
	`(Typ)`
	`sizeof`
	`_Alignof`
		Präfix-Inkrement
		Präfix-Dekrement
		Vorzeichen
		Vorzeichen
		Logisches NICHT
		Bitweises NICHT
		Adresse
		Zeigerdereferenzierung
		Typumwandlung
		Speichergröße
		Speicherausrichtung$_{(C11)}$
			Rechts nach links

3	`*`
	`/`
	`%`
		Multiplikation
		Division
		Modulo
			Links nach rechts

Tabelle A.1 Operatoren von C und ihre Assoziativität

Rang	Operator	Beschreibung	Richtung
4	`+` `-`	Addition Subtraktion	Links nach rechts
5	`<<` `>>`	Links-Shift Rechts-Shift	Links nach rechts

6	`<`
	`<=`
	`>`
	`>=`
		Kleiner
		Kleiner-gleich
		Größer
		Größer-gleich
			Links nach rechts

7	`==` `!=`	Gleich Ungleich	Links nach rechts
8	`&`	Bitweises UND	Links nach rechts
9	`^`	Bitweises exklusives ODER	Links nach rechts
10	`|`	Bitweises ODER	Links nach rechts
11	`&&`	Logisches UND	Links nach rechts
12	`||`	Logisches ODER	Links nach rechts
13	`?:`	Bedingung	Rechts nach links
14	`=` `*=, /=, +=, -=, &=, ^=, |=, <<=, >>=`	Zuweisung Zusammengesetzte Zuweisung	Rechts nach links
15	`,`	Komma-Operator	Links nach rechts

Tabelle A.1 Operatoren von C und ihre Assoziativität (Forts.)

A.2 Reservierte Schlüsselwörter in C

Schlüsselwörter sind Wörter mit einer vorgegebenen Bedeutung in C. Sie dürfen nicht anderweitig verwendet werden. So dürfen Sie beispielsweise

keine Variable mit dem Bezeichner `int` verwenden, da es auch einen Basisdatentyp mit diesem Namen gibt. Der Compiler würde sich ohnehin darüber beschweren. In der folgenden Tabelle finden Sie einen Überblick zu den reservierten Schlüsselwörtern in C.

auto	break	case	char
const	continue	default	do
double	else	enum	extern
float	for	goto	if
int	long	register	return
short	signed	sizeof	unsigned
void	volatile	while	
C99			
inline	restrict	_Bool	_Complex
_Imaginary			
C11			
_Alignas	_Alignof	_Atomic	_Generic
_Noreturn	_Static_assert	_Thread_local	

Tabelle A.2 Reservierte Schlüsselwörter in C

A.3 Headerdateien der Standardbibliothek

In der folgenden Tabelle finden Sie einen Überblick über die verschiedenen Headerdateien, die Ihnen heute mit C11 zur Verfügung stehen. Da es wohl immer noch Compiler gibt, die den C11-Standard nur teilweise bis gar nicht implementiert haben, wird auch markiert, ab welchem Standard die entsprechende Headerdatei der Standardbibliothek vorhanden ist.

Headerdatei	Standard	Bedeutung
<assert.h>	C89/C90	Assertions; Fehlersuche
<complex.h>	C99	Komplexe Zahlenarithmetik
<ctype.h>	C89/C90	Test auf bestimmte Zeichentypen
<errno.h>	C89/C90	Makros mit Fehlercodes
<fenv.h	C99	Einstellungen für die Gleitkomma-berechnungen
<float.h>	C89/C90	Limits für Gleitkommazahlen
<inttypes.h>	C99	Konvertierungsfunktionen für Ganzzahl-typen
<iso646.h>	C95/NA1	Alternative Schreibweise für logische und bitweise Operatoren
<limits.h>	C89/C90	Größe eingebauter Typen
<locale.h>	C89/C90	Einstellungen des Gebietsschemas
<math.h>	C89/C90	Mathematische Funktionen
<setjmp.h>	C89/C90	Nichtlokale Sprünge
<signal.h>	C89/C90	Signalverarbeitung
<stdalign.h>	C11	Makros für Speicherausrichtung
<stdarg.h>	C89/C90	Variable Anzahl von Argumenten
<stdatomic.h>	C11	Typen für atomare Operationen für Threads
<stdbool.h>	C99	Boolesche Variablen
<stddef.h>	C89/C90	Zusätzliche Typendefinitionen

Tabelle A.3 C-Standardbibliothek-Headerdateien

Headerdatei	Standard	Bedeutung
<stdint.h>	C99	Ganzzahltypen mit fester Breite
<stdio.h>	C89/C90	Ein-/Ausgabe
<stdlib.h>	C89/C90	Allgemeine Standardfunktionen
<stdnoreturn.h>	C11	Definition des Noreturn-Makros
<string.h>	C89/C90	Funktionen für Zeichenketten
<tgmath.h>	C99	Typgenerische Makros für mathematische Funktionen
<threads.h>	C11	Unterstützung von Multithreads
<time.h>	C89/C90	Datum und Uhrzeit
<uchar.h>	C11	Unterstützung von Unicode-Zeichen (UTF-16- und UTF-32-kodiert)
<wchar.h>	C95/NA1	Unterstützung für Unicode-Zeichen
<wctype.h>	C95/NA1	Wie <ctype.h>, nur für Unicode

Tabelle A.3 C-Standardbibliothek-Headerdateien (Forts.)

A.4 Kommandozeilenargumente

Beim Start-up des Programms wird die `main()`-Funktion aufgerufen. Darauf wurde ja bereits in Abschnitt 7.9, »main-Funktion«, eingegangen. Neben der Möglichkeit, die `main`-Funktion wie folgt zu definieren:

```
int main(void) { /* ... */ }
```

gibt es auch noch die Möglichkeit, (Kommandozeilen-)Argumente beim Programmstart an die `main`-Funktion zu übergeben. Für diesen Fall können Sie `main()` so definieren, wie im Folgenden gezeigt.

Damit Sie einem Programm beim Start Argumente übergeben können, wird eine parametrisierte Hauptfunktion benötigt. Ihre Syntax sieht so aus:

```
int main(int argc, char *argv[]) { /* ... */ }
```

Die Hauptfunktion `main()` besitzt hier zwei Parameter mit den Namen `argc` und `argv`. Die Bezeichner `argc` und `argv` dieser Parameter sind so nicht vorgeschrieben, und Sie können hierfür auch einen beliebigen gültigen Bezeichner verwenden. In der Praxis werden allerdings die beiden Bezeichner `argc` (für *argument counter*) und `argv` (für *argument vector*) meistens mit diesem Namen verwendet, weil dann auch andere Programmierer, die den Code lesen müssen, schnell erkennen, worum es sich dabei handelt.

Der erste Parameter `argc` beinhaltet die Anzahl Argumente, die dem Programm beim Start übergeben wurden. Dabei handelt es sich um einen Integerwert. Im zweiten Parameter `argv` stehen die einzelnen Argumente. Diese werden in einem Array von Zeigern auf `char` gespeichert. Folgendes Beispiel demonstriert dies:

```
/* argument.c */
#include <stdio.h>
#include <stdlib.h>

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

Das Listing wurde z. B. unter dem Namen *argument.c* gespeichert und anschließend übersetzt. Wenn Sie das Programm starten, wird auf dem Bildschirm meistens der Programmname ausgegeben:

```
$ ./argument
argv[0] = ./argument
```

Ob `argv[0]` der Programmname ist, hängt allerdings auch von der Programmumgebung ab. Es ist daher auch möglich, dass `argv[0]` einfach nur ein Leerstring "" ist.

Starten Sie das Programm jetzt nochmals mit folgender Eingabe von der Kommandozeile (*argument* sei wieder der Programmname):

```
$ ./argument Hallo Welt
argv[0] = ./argument
argv[1] = Hallo
argv[2] = Welt
```

Die einzelnen Argumente, die dem Programm übergeben werden, müssen immer durch mindestens ein Leerzeichen getrennt sein. Wenn Sie z. B. Folgendes eingeben:

```
$ ./argument HalloWelt
argv[0] = argument
argv[1] = HalloWelt
```

werden nur noch zwei Argumente ausgegeben, weil zwischen *Hallo* und *Welt* kein Leerzeichen mehr war.

Der Parameter `int argc` zählt die Anzahl der Strings, die dem Programm beim Aufruf mitgegeben wurden. Dazu ein Beispiel:

```
/* arg_counter.c */
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[]) {
  printf("Insgesamt %d Argumente\n", argc-1);
  printf("Letztes Argument: %s\n", argv[argc-1]);
  return EXIT_SUCCESS;
}
```

Bei diesem Beispiel werden die Anzahl der Argumente und das letzte Argument ausgegeben. Es kann nützlich sein, die Anzahl der Argumente zu kennen, wenn Sie ein Programm erstellen, das eine bestimmte Anzahl von Argumenten erfordert.

Falls Sie ein Programm schreiben, das unbedingt eine bestimmte Anzahl Argumente beim Startup benötigt oder erwartet, dann sollten Sie immer die Anzahl der Argumente überprüfen, damit es nicht zu einer Fehlfunktion kommt. Das folgende Beispiel demonstriert dies:

```
/* min_args.c */
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[]) {
  if( argc < 2 ) {
    fprintf(stderr, "Mind. ein Argument erwartet\n");
    return EXIT_FAILURE;
  }
  else {
    printf("Die eingegebenen Argumente:\n");
    for(int i = 1; i < argc; i++ ) {
      printf("\t %d : %s\n", i, argv[i]);
    }
  }
  return EXIT_SUCCESS;
}
```

A.5 Weiterführende Ressourcen

- N1507 Committee Draft (ISO/IEC 9899:201x):
  http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1570.pdf
- Gute Online-Referenz der Standardbibliothek:
  http://en.cppreference.com/w/c
- Erweiterung für die neuen Zeichentypen (Unicode-Support):
  http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1040.pdf

- Neue Schnittstellen für die Speicherbereichsüberschreitung (*bound-checking interface*): *http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1225.pdf*
- Erweiterung zu parallelen Programmierung: *http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1966.pdf*
- Regeln zur sichereren Programmierung: *http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1624.pdf*
- Vermeiden von Sicherheitslücken bei der Programmierung: *http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1980.pdf*

A.6 Schlusswort

Hier angekommen gilt: Nach dem Buch ist vor dem Buch, und es gibt in C noch sehr vieles zu erlernen. Das Ziel dieses Grundkurs-Taschenbuchs war es, Ihnen den Sprachkern von C näher zu bringen. Aufbauend darauf haben Sie es jetzt selbst in der Hand, tiefer in die Welt von C einzusteigen.

Definitiv ist es dabei immer empfehlenswert, auch den C11-Standard (*http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1570.pdf*) zu Rate zu ziehen.

Anhang B

Lösungen der Übungsaufgaben

B.1 Antworten und Lösungen zum Kapitel 2

1. Hier die Lösung der gültigen und ungültigen Bezeichner:

```
anzahlPreise<30;      // Fehler <
_#Preise_kleiner_30;  // Fehler #
_groesster_Wert;      // nicht empfehlenswert wegen _
groesster_Wert;       // OK
größter_Wert;         // erlaubt gemäß C11; von wenigen Compilern
                      // unterstützt
```

2. Folgende drei Fehler waren im Programm enthalten:

```
00  // kap002/loesung001.c
01  #include <stdio.h>

02  int main(void) {        // main, nicht Main!!!
03     printf("Ei-Pod\n");  // " anstatt ' (Zeichenkette)
04     return 0;            // Semikolon am Ende fehlte
05  }
```

3. In dem Beispiel wurde am Anfang vergessen, die Headerdatei `stdio.h` zu inkludieren. Es fehlt also die folgende Zeile:

```
// kap002/loesung002.c
#include <stdio.h>
...
```

4. Hier eine mögliche Lösung der Aufgabe:

```
00  // kap002/loesung003.c
01  #include <stdio.h>

02  int main(void) {
03     printf("\tJ\n\tu\n\ts\n\tt ");
04     printf("  for  F\n\t\tu\n\t\tn\n");
05     return 0;
06  }
```

B.2 Antworten und Lösungen zum Kapitel 3

1. `signed char`, `short`, `int`, `long` und `long long`.
2. Mit dem Schlüsselwort `unsigned` kann eine ganzzahlige Variable ohne Vorzeichen vereinbart werden.
3. Für Ganzzahltypen mit fester Bitbreite gibt es seit C99 verschiedene plattformunabhängige Typen, die in der Headerdatei `<stdint.h>` definiert sind.
4. Der grundlegende Datentyp für Zeichen lautet `char`.
5. Eine einfache Musterlösung mit dem `sizeof`-Operator sieht wie folgt aus:

```
00  // kap003/loesung001.c
01  #include <stdio.h>

02  int main(void) {
03    printf("int: %zu Bytes\n", sizeof(int));
04    printf("long long: %zu Bytes\n", sizeof(long long));
05    return 0;
06  }
```

6. Die implementierungsabhängigen Wertebereiche für Integertypen sind in der Headerdatei `<limits.h>` und die für Gleitkommazahlen in der Headerdatei `<float.h>` definiert.
7. Mit dem Qualifikator `const` können Sie nicht mehr veränderbare Variablen (Konstanten) definieren.

B.3 Antworten und Lösungen zum Kapitel 4

1. Hier wurde der Adressoperator bei `scanf` vergessen:

```
// kap004/loesung001.c
#include <stdio.h>

int main(void) {
  int iVar = 0;
  printf("Bitte eine Ganzzahl eingeben: ");
```

```
  int check = scanf("%d", &iVar); // mit Adressoperator

  if( check != 1 ) {
    printf("Fehler bei scanf...\n");
    return 1;  // Programm beenden
  }
  printf("%d Wert(e) eingelesen; ", check);
  printf("der eingegebene Wert lautet: %d\n", iVar);
  return 0;
}
```

2. Eine einfache Musterlösung könnte wie folgt aussehen:

```
// kap004/loesung002.c
#include <stdio.h>

int main(void) {
  double cel = 0.0;
  printf("Temperatur in Celsius: ");
  int check = scanf("%lf", &cel );
  if( check != 1 ) {
    printf("Fehler bei der Eingabe ...\n");
    return 1;
  }
  double kelvin = cel + 273.15;
  double fahrenheit = (cel*9)/5+32;
  printf("%lf Grad Celsius sind ...\n", cel);
  printf("%lf Kelvin\n", kelvin);
  printf("%lf Grad Fahrenheit\n", fahrenheit);
  return 0;
}
```

Das Programm sieht bei der Ausführung folgendermaßen aus:

```
Temperatur in Celsius: 30
30.000000 Grad Celsius sind ...
303.150000 Kelvin
86.000000 Grad Fahrenheit
```

3. Die Ausgabe wurde als Kommentar hinzugefügt:

```
int i = 1;
printf("i = %d\n", i--);     // i = 1
printf("i = %d\n", ++i);     // i = 1
printf("i = %d\n", i++);     // i = 1
printf("i = %d\n", ++i);     // i = 3
```

4. Bei der *impliziten Typenumwandlung* nimmt der Compiler automatisch die Konvertierung von einem Datentyp zum anderen nach einem Regelwerk vor.
5. Mit der *expliziten Typenumwandlung* kann der Programmierer selbst eine Umwandlung von einem zum anderen Datentyp erzwingen. Eine *explizite Umwandlung* sollten Sie immer vornehmen, wenn bei einer Umwandlung Speicherinformationen verloren gehen könnten (beispielsweise zwischen Gleitpunktzahlen und Ganzzahlen).
6. Die *explizite Umwandlung* wird mithilfe des *Cast-Operators* durchgeführt:

```
(typ) ausdruck
```

   Dabei wird immer zuerst `ausdruck` ausgewertet, ehe der in `ausdruck` zurückgegebene Wert in den Datentyp `typ` konvertiert wird.

B.4 Antworten und Lösungen zum Kapitel 5

1. Wird die Bedingung der `if`-Anweisung erfüllt (ist wahr; true), wird ein Wert ungleich 0 zurückgeliefert. Ansonsten wird bei einer falschen Bedingung (ist unwahr; false) gleich 0 zurückgegeben.
2. Die alternative, aber optionale Verzweigung einer `if`-Anweisung wird mit `else` eingeleitet. Sie wird ausgeführt, wenn die Bedingung von `if` unwahr (also gleich 0) ist.
3. Sie können entweder mehrere `if`-Anweisungen verketten (mit `else if`) oder eine Fallunterscheidung mit `switch` vornehmen.
4. `break` dient als Ausstiegspunkt aus dem `switch`-Konstrukt. Wird kein `break` am Ende einer `case`-Marke verwendet und eine entsprechende Marke angesprungen, werden sämtliche Anweisungen des `switch`-Kon-

struktes dahinter abgearbeitet, bis zum Ende des `switch`-Rumpfes oder bis zum nächsten `break`. Dies kann allerdings durchaus gewollt sein und ist daher rein syntaktisch kein Fehler.

5. Findet keine Übereinstimmung bei den `case`-Marken statt, können Sie eine `default`-Marke setzen, zu der verzweigt wird.
6. Es gibt den logischen UND-Operator (&&), den logischen ODER-Operator (||) und den logischen NICHT-Operator (!). In der Praxis werden diese Operatoren verwendet, um mehrere Ausdrücke miteinander zu einer Bedingung zu verknüpfen. Damit lassen sich beispielsweise Verzweigungen ausführen, bei denen zwei oder mehrere Bedingungen gleichzeitig zutreffen sollen (wahr; ungleich 0) oder auch nicht zutreffen sollen (unwahr, gleich 0).
7. Folgende Werte haben die logischen Verknüpfungen ergeben:

```
01 logo1 = 1  // wahr
02 logo2 = 1  // wahr
03 logo3 = 0  // unwahr
04 logo4 = 0  // unwahr
05 logo5 = 1  // wahr
```

8. Hierzu eine mögliche Musterlösung:

```
00 // kap005/loesung001.c
01 #include <stdio.h>

02 int main(void) {
03   int ival = 0;
04   printf("Zahl zwischen 1-100 eingeben: ");
05   if( scanf("%d", &ival) != 1 ) {
06     printf("Fehler bei der Eingabe...\n");
07     return 1;
08   }
09   if( (ival >= 1) && (ival <= 100) ) {
10     if( ival % 2 ) {
11       printf("Die Zahl ist ungerade\n");
12     }
13     else {
14       printf("Die Zahl ist gerade\n");
```

```
15      }
16    }
17    else {
18      printf("Die Zahl war nicht 1-100!\n");
19    }
20    return 0;
21  }
```

9. Hierzu eine mögliche Musterlösung:

```
00  // kap005/loesung002.c
01  #include <stdio.h>

02  int main(void) {
03    int work = 0;
04    printf("-1- PC 1 hochfahren\n");
05    printf("-2- PC 2 hochfahren\n");
06    printf("-3- Drucker einschalten\n");
07    printf("-4- Kaffee machen\n");
08    printf("-5- Feierabend machen\n");
09    printf("Was wollen Sie tun: ");
10    if( scanf("%d", &work) != 1 ) {
11      printf("Fehler bei der Eingabe...\n");
12      return 1;
13    }
14    switch( work ) {
15      case 1 : printf("PC 1 wird hochgefahren\n");
16               break;
17      case 2 : printf("PC 2 wird hochgefahren\n");
18               break;
19      case 3 : printf("Drucker wird eingeschaltet\n");
20               break;
21      case 4 : printf("Kaffee wird gemacht\n");
22               break;
23      case 5 : printf("Gute Nacht\n");
24               break;
25      default: printf("Falsche Eingabe!\n");
```

```
26    }
27    return 0;
28  }
```

B.5 Antworten und Lösungen zum Kapitel 6

1. Schleifen sind Kontrollstrukturen, mit denen Anweisungen, die meistens in einem Anweisungsblock zusammengefasst sind, so oft wiederholt werden, bis eine bestimmte Abbruchbedingung erreicht wird.
2. In C stehen die Zählschleife `for`, die kopfgesteuerte Schleife `while` und die fußgesteuerte Schleife `do-while` zur Verfügung.
3. Muss ein Block von Anweisungen mindestens einmal ausgeführt werden, sollten Sie die `do-while`-Schleife verwenden. Mithilfe der `do-while`-Schleife wird die Abbruchbedingung nämlich erst am Ende des Schleifenrumpfes überprüft. Alternativ kann auch mithilfe der `while`- oder `for`-Schleife und der `break`-Anweisung eine solche einmal auszuführende Schleife nachgebildet werden.
4. Mit dem Schlüsselwort `break` können Sie eine Schleife vorzeitig verlassen, und mit dem Schlüsselwort `continue` können Sie den aktuellen Schleifendurchlauf vorzeitig beenden.
5. Die Schleife gibt gar nichts aus und wird nach der ersten Überprüfung wieder abgebrochen. In diesem Beispiel wurde eine falsche Abbruchbedingung in der Schleife definiert. Die Schleife wird ausgeführt, solange der Wert von `ival` größer als 10 ist, und das trifft hier gleich von Beginn an nicht zu. Die Schleifenbedingung müsste `while` lauten, solange der Wert von `ival` **kleiner** als 10 ist, also:

   ```
   While ( ival < 10 ) {
     ...
   }
   ```

6. Dies ist ein typischer, aber nicht sofort ersichtlicher logischer Fehler. Bei den Gleitpunktzahlen gibt es das Problem, dass sie nicht exakt dargestellt werden können. In diesem Beispiel sollten Sie daher unbedingt auf kleiner-gleich statt auf ungleich prüfen. Die bessere Lösung wäre also:

```
for(float fval = 0.0f; fval <= 1.0f; fval+=0.1f) {
   printf("%f\n", fval);
}
```

7. Nach dem ersten Schleifendurchlauf wird in diesem Beispiel noch 0 ausgegeben. Im nächsten Schleifendurchlauf, wenn die Schleifenvariable den Wert 1 hat, trifft die `if`-Verzweigung zu, dass eine Division von `ival` durch 2 einen Rest ergibt. Deshalb wird mit `continue` wieder zur `while`-Schleife hochgesprungen. Dort hat sich die Bedingung natürlich nicht geändert, und im nächsten Schleifendurchlauf bleibt man erneut in der `if`-Verzweigung hängen. Sie haben quasi in diesem kleinen Bereich eine Endlosschleife. Wegen des Schlüsselworts `continue` wird das Hochzählen der Schleifenvariablen `ival++` nicht mehr erreicht. Dies ist aber von enormer Bedeutung für die Beendigung der Schleife. Besser wäre es also, die Schleifenvariable auch vor der `continue`-Anweisung hochzuzählen. Noch besser wäre es aber, gleich eine `for`-Schleife zu verwenden:

```
int ival=0;
for(ival = 0; ival < 20; ival++) {
  if(ival % 2) {
     continue;
  }
  printf("%d\n", ival);
}
```

8. Hierzu eine Musterlösung:

```
00  // kap006/loesung001.c
01  #include <stdio.h>

02  int main(void) {
03    double geld=0.0, zs_tmp=0.0, zs=0.0;
04    unsigned int jahre=0;
05    printf("Welchen Geldbetrag wollen Sie einzahlen: ");
06    if( scanf("%lf", &geld) != 1 ) {
07      printf("Fehler bei der Eingabe...\n");
08      return 1;
09    }
```

```
10    printf("Wie viel Zinsen bekommen Sie hierbei(%%): ");
11    if( scanf("%lf", &zs_tmp) != 1 ) {
12      printf("Fehler bei der Eingabe...\n");
13      return 1;
14    }
15    zs = (zs_tmp / 100) + 1.0f;
16    printf("Wie viele Jahre wollen Sie sparen      : ");
17    if( scanf("%u", &jahre) != 1 ) {
18      printf("Fehler bei der Eingabe...\n");
19      return 1;
20    }
21    for(unsigned int i = 0; i < jahre; geld=geld*zs, i++ ) {
22      printf("%2u. Jahr: %.2lf\n",i+1, geld);
23    }
24    return 0;
25  }
```

Das Programm bei der Ausführung:

```
Welchen Geldbetrag wollen Sie einzahlen: 100000
Wie viel Zinsen bekommen Sie hierbei(%): 1.2
Wie viele Jahre wollen Sie sparen      : 10
 1. Jahr: 100000.00
 2. Jahr: 101200.00
 3. Jahr: 102414.40
 4. Jahr: 103643.37
 5. Jahr: 104887.09
 6. Jahr: 106145.74
 7. Jahr: 107419.49
 8. Jahr: 108708.52
 9. Jahr: 110013.02
10. Jahr: 111333.18
```

B.6 Antworten und Lösungen zum Kapitel 7

1. Bei einer Vorausdeklaration wird nur der Funktionskopf mit abschließendem Semikolon vor dem Aufruf deklariert (ein Funktionsprototyp).

Dem Compiler reicht es hierbei aus, nur den Rückgabetyp, den Bezeichner und die Typen der formalen Parameter zu kennen. Bei einer Deklaration müssen Sie nicht den Anweisungsblock der Funktion definieren. Durch eine Vorwärts-Deklaration kann sich die Definition (wo Speicherplatz für die Funktion reserviert wird) irgendwo im Programm befinden.

2. Mit call-by-value werden Daten als Argumente an eine Funktion übergeben. Diese Daten werden dabei kopiert. Somit besteht keine Verbindung mehr zwischen dem Aufrufer der Daten und der Funktion. Eine Änderung der Daten in der Funktion hat damit keine Auswirkung auf die Daten des Aufrufers. Der Nachteil bei diesem Verfahren ist, dass bei umfangreichen Daten, die durch das Kopieren an die Funktion übergeben werden, das Laufzeitverhalten des Programms gebremst wird.
3. Einen Wert an den Aufrufer können Sie mit der `return`-Anweisung zurückgeben. Hierbei sollten Sie beachten, dass der mit `return` zurückgegebene Wert mit dem Rückgabewert der Funktionsdefinition übereinstimmt.
4. Für gewöhnlich kümmert sich das Stacksegment um die Funktionen. Wird eine Funktion aufgerufen, wird auf dem Stack ein Datenblock angelegt, in dem die Rücksprungadresse zum Aufrufen, die Funktionsparameter und die lokalen Variablen der Funktion gespeichert werden. Wird die Funktion wieder verlassen, werden diese Daten freigegeben und sind unwiderruflich verloren.
5. Eine Rekursion bezeichnet eine Funktion, die sich immer wieder selbst aufruft. Wichtig bei solchen sich selbst aufrufenden Funktionen ist eine Abbruchbedingung, die den Vorgang beendet. Nach Möglichkeit sollten Sie allerdings, wegen des enormen Aufwandes auf dem Stacksegment, auf Rekursionen verzichten und einen iterativen Lösungsansatz verwenden.
6. Eine lokale Variable wird immer innerhalb eines Anweisungsblocks definiert und ist nach außen hin nicht sichtbar. Wird der Anweisungsblock beendet, verliert auch die lokale Variable ihren Wert. Globale Variablen hingegen werden außerhalb der Anweisungsblöcke und Funktionen definiert und sind im kompletten Programm sichtbar. Außerdem werden globale Variablen, im Gegensatz zu den lokalen Variablen, bei der Definition initialisiert.

7. Verwenden Sie den Speicherklassen-Spezifizierer `static`, ist eine Variable oder eine Funktion nur noch in der aktuellen Quelldatei sichtbar.
8. Entweder verwenden Sie eine globale Variable (schlechtere Lösung) oder Sie kennzeichnen eine Variable im Funktionsblock mit dem Spezifizierer `static`.
9. In dem Beispiel wurde die Vorausdeklaration der Funktion `multi()` vergessen:

```
00 // kap007/loesung001.c
01 #include <stdio.h>
02 #include <stdlib.h>

03 float multi(float);

04 int main(void) {
05   float fval = multi(3.33);
06   printf("%.2f\n", fval);
07   return EXIT_SUCCESS;
08 }

09 float multi(float f) {
10   return (f*f);
11 }
```

10. In diesem Beispiel wurde vergessen, das Ergebnis der Berechnung mittels `return` zurückzugeben:

```
00 // kap007/loesung002.c
01 #include <stdio.h>
02 #include <stdlib.h>

03 float volumen_Rect(float l, float b, float h) {
04   float volumen = l*b*h;
05   return volumen;
06   // oder gleich: return l*b*h;
07 }

08 int main(void) {
```

```
09    float vol = volumen_Rect(10.0, 10.0, 12.5);
10    printf("Volumen: %f\n", vol);
11    return EXIT_SUCCESS;
12  }
```

11. Hierzu eine mögliche Lösung, die Fakultät ohne eine Rekursion zu berechnen:

```
00  // kap007/loesung003.c
01  #include <stdio.h>

02  long fakul( long n ) {
03    long val = n;
04    while(--val ) {
05      n*=val;
06    }
07    return n;
08  }

09  int main(void) {
10    printf("Fakultät von 6: %ld\n", fakul(6));
11    printf("Fakultät von 8: %ld\n", fakul(8));
12    return 0;
13  }
```

B.7 Antworten und Lösungen zum Kapitel 8

1. Ist die Datei in einfachen Anführungszeichen eingeschlossen (beispielsweise `"datei.h"`), wird sie im aktuellen Verzeichnis (bzw. im eventuell angegebenen relativen oder absoluten Pfad) gesucht. Wird sie dort nicht gefunden, wird sie im Systemverzeichnis gesucht, wo sich die Headerdateien im Allgemeinen befinden. Das Verzeichnis zu den Headern kann aber auch mit speziellen Compiler-Flags (abhängig vom Compiler) angegeben werden. Wird die Datei hingegen mit spitzen Klammern eingebunden (beispielsweise `<datei.h>`), wird nur im Systemverzeichnis des Compilers nach den Headerdateien gesucht.

2. Mit der `define`-Direktive können Sie einen Text angeben, der vor der Übersetzung des Compilers vom Präprozessor durch den dahinterstehenden Ersetzungstext im Quellcode ersetzt wird. Dies wird beispielsweise sowohl für die Definition von symbolischen Konstanten als auch für die Definition von Makros (mit Parametern) verwendet.
3. Mit der `undef`-Direktive können Sie definierte symbolische Konstanten oder Makros jederzeit wieder aufheben. Alternativ können Sie auch nur die `define`-Direktive entfernen. Beachten Sie allerdings: Ab der Stelle, an der Sie mit `undef` eine Konstante oder ein Makro aufheben, beschwert sich der Compiler, dass ein ungültiger Bezeichner verwendet wird, wenn Sie diesen nach der Zeile mit `undef` trotzdem noch verwenden.
4. Die Entscheidung sollte in diesem Fall auf die Readonly-Variable mit dem Schlüsselwort `const` fallen. Würden Sie die symbolische Konstante mit `define` verwenden, würde nur an jeder Stelle im Programm, an der `VAL` verwendet wird, eine textuelle Ersetzung durchgeführt. Während der Programmausführung bedeutet das, dass an dieser Stelle jedes Mal `VAL` neu berechnet wird. Bei einer `const`-Variablen hingegen findet die Berechnung nur einmal im Programm statt.
5. Mit einer bedingten Kompilierung können Sie die Übersetzung eines Programms von Präprozessor-Anweisungen und -Bedingungen abhängig machen. So können Sie beispielsweise entscheiden, auf welchem System ein Quellcode übersetzt wird und welche Headerdateien oder sonstiger Code inkludiert werden. Zusätzlich vermeiden Sie das Problem, dass ein Header mehrfach inkludiert wird. Eine weitere interessante Lösung für die bedingte Kompilierung dürfte das Debuggen sein.
6. Wenn Sie die Parameter im Ersatztext des Makros in Zeile **(03)** in Klammern setzen, sind Sie immer auf der sicheren Seite.

```
00  // kap008/loesung001.c
01  #include <stdio.h>
02  #include <stdlib.h>
03  #define MULTI(a, b) ((a)*(b))

04  int main(void) {
05    int val1 = 10, val2 = 20;
```

```
06    printf("Multiplikation = %d\n", MULTI(val1, val2-10));
07    return EXIT_SUCCESS;
08  }
```

7. Die Schleife wird fünfmal durchlaufen. Die erste symbolische Konstante in Zeile **(03)** wird in Zeile **(06)** mit der `undef`-Direktive aufgehoben und in Zeile **(07)** auf den Wert 5 gesetzt. In Zeile **(08)** wird dieser Text dann vom Präprozessor ersetzt, sodass die Zeilen **(09)** und **(10)** keinen Effekt mehr auf die Schleife haben. Hinter Zeile **(10)** hat allerdings die symbolische Konstante den Wert 20.

8. Die Lösung ist einfacher, als Sie vielleicht vermutet haben. Hier ein möglicher Lösungsansatz:

```
00  // kap008/mysyntax.h
01  #ifndef MYSYNTAX_H
02  #define MYSYNTAX_H
03  #include <stdio.h>
04  #include <stdlib.h>

05  #define MAIN   int main(void)
06  #define OPEN   {
07  #define CLOSE  }
08  #define END    return EXIT_SUCCESS;
09  #define WRITE printf(
10  #define WRITE_ );

11  #endif
```

9. Hier ein einfacher Lösungsansatz zu den beiden Makros, die mit dem ternären Operator realisiert wurden.

```
00  // kap008/loesung002.c
01  #include <stdio.h>
02  #include <stdlib.h>
03  #define MAX(a, b)  (((a) > (b)) ? (a) : (b))
04  #define MIN(a, b)  (((a) < (b)) ? (a) : (b))

05  int main(void) {
06    int val1 = 20, val2 = 30;
```

```
07    printf("Max. Wert: %d\n", MAX(val1, val2) );
08    printf("Min. Wert: %d\n", MIN(val1, val2) );
09    return EXIT_SUCCESS;
10  }
```

10. Auch hier ist die Lösung nicht schwer, weil alle Daten als vordefinierte Standardmakros enthalten sind. Ein möglicher Lösungsansatz lautet:

```
00  // kap008/loesung003.c
01  #include <stdio.h>
02  #include <stdlib.h>
03  #define DEBUG_TIME printf("%s / %s\n", __DATE__,__TIME__);
04  #define DEBUG_LINE printf("%d / %s\n", __LINE__,__FILE__);
05  #define DEBUG_ALL { DEBUG_TIME DEBUG_LINE }

06  int main(void) {
07    DEBUG_ALL;
08    return EXIT_SUCCESS;
09  }
```

B.8 Antworten und Lösungen zum Kapitel 9

1. Arrays sind zusammengesetzte Datenstrukturen, in denen sich mehrere Werte eines bestimmten Typs als Folge abspeichern und bearbeiten lassen. Der Zugriff auf die Daten erfolgt mithilfe des Indizierungsoperators `[]` über einen Index.
2. Zunächst einmal sind Strings nichts anderes als Arrays vom Datentyp `char` (bzw. `wchar_t`). Allerdings sind die `char`-Arrays die einzige Möglichkeit, eine Folge von Zeichen, also einen String, auch Zeichenkette genannt, auszugeben. Einen eigenen Datentyp für Strings gibt es in C nicht. Wichtig bei einem `char`-Array: Damit Sie diesen auch wirklich als echten String verwenden können, müssen Sie ihn mit einem Null-Zeichen oder auch Stringende-Zeichen `'\0'` abschließen. Demnach muss ein String immer um ein Zeichen länger sein als die Anzahl der relevanten Zeichen, damit das Stringende-Zeichen am Ende auch noch Platz hat.
3. Die größte Gefahr geht davon aus, dass Sie mehr Daten in ein Array oder einen String schreiben, als Speicherplatz dafür vorhanden ist.

Haben Sie beispielsweise ein Array mit 10 Elementen und schreiben 11 Elemente in das Array, weil Sie den Indexbereich versehentlich überschreiten, haben Sie eine Speicherüberschreitung. Häufig verabschiedet sich das Programm dann mit einem Speicherzugriffsfehler. Ein solcher Fehler kann aber nur schwer gefunden werden und zu falschen Daten führen. In der Netzwerkprogrammierung kann dies ganz böse Folgen haben, weil sich solche Fehler dann ausnutzen lassen, um Schaden auf dem jeweiligen Rechner anzurichten.

4. Ein beliebter Anfängerfehler ist es zu vergessen, dass das erste Element eines Arrays bzw. Strings immer mit der Indexnummer 0 beginnt und das letzte Element immer die Indexnummer `N-1` (`N` ist die angegebene Array-Größe) besitzt. Daher wird hier oft ein Überlauf erzeugt, weil auf die Indexnummer `N` anstatt auf `N-1` zugegriffen wird. Bei Strings wird häufig vergessen, dass noch Platz für das Stringende-Zeichen vorhanden sein muss.

5. In der `for`-Schleife der Zeile **(06)** wurden gleich zwei Fehler gemacht. Da `i` mit `MAX` initialisiert wurde, wurde dem Array mit dem Index 10 ein Wert übergeben. Somit wäre dies ein Array-Überlauf, weil die Indexnummern nur von 0 bis 9 gehen. Des Weiteren würde durch die Überprüfung, ob `i` größer als 0 ist, das Arrayelement mit dem Index 0 nicht initialisiert. In der `for`-Schleife der Zeile **(06)** muss daher `MAX-1` an `i` übergeben werden, und die Schleifenbedingung sollte auf größer-gleich 0 geprüft werden:

```
06    for(int i = MAX-1; I >= 0; i--) {
```

6. Dass das `char`-Array `v` in der Zeile **(05)** nicht mit `\0` abgeschlossen wird, ist nicht der Fehler. Der Fehler wird erst in Zeile **(10)** des Programms gemacht. Dort wird das `char`-Array als String behandelt. Das sollte man vermeiden, wenn kein Stringende-Zeichen vorhanden ist. Sie haben folgende zwei Möglichkeiten, damit `v` als korrekter String angesehen werden darf:

```
char v[6] = { 'A', 'E', 'I', 'O', 'U', '\0' };
char v[6] = { "AEIOU" };
```

7. Hier eine mögliche Musterlösung der Aufgabe:

```
00  // kap009/loesung001.c
01  #include <stdio.h>
02  #include <stdlib.h>
```

```
03  #include <string.h>

04  int main(void) {
05    int iarr[]    = { 2, 4, 6, 4, 2, 4, 5, 6, 7 };
06    double darr[] = { 3.3, 4.4, 2.3, 5.8, 7.7 };
07    char str[]    = { "Hallo Welt"};

08    printf("iarr: %zu Bytes\n", sizeof(iarr));
09    printf("iarr: %zu Elemente\n", sizeof(iarr)/sizeof(int));
10    printf("darr: %zu Bytes\n", sizeof(darr));
11    printf("darr: %zu Elemente\n",sizeof(darr)/sizeof(double));
12    printf("str : %zu Bytes\n", sizeof(str));
13    printf("str : %zu Elemente (sizeof)\n",sizeof(str)/sizeof(char));
14    printf("str : %zu Elemente (strlen)\n", strlen(str)+1);
15    return EXIT_SUCCESS;
16  }
```

Das Programm bei der Ausführung:

```
iarr: 36 Bytes
iarr: 9 Elemente
darr: 40 Bytes
darr: 5 Elemente
str : 11 Bytes
str : 11 Elemente (sizeof)
str : 11 Elemente (strlen)
```

8. Hierzu eine Musterlösung:

```
00  // kap009/loesung002.c
01  #include <stdio.h>
02  #include <stdlib.h>
03  #define EQUAL -1
04  #define DIFFSIZE -2

05  int vergleich( int arr1[], int n1, int arr2[], int n2 ) {
06    if(n1 != n2) {
07      return DIFFSIZE; // Arrays unterschiedlich lang
08    }
```

```
09    for(int i=0; i < n1; i++) {
10      if( arr1[i] != arr2[i] ) {
11        return i; // Indexnummer mit Unterschied
12      }
13    }
14    return EQUAL; // Beide Arrays sind identisch
15  }

16  int main(void) {
17    int iarr1[] = { 2, 1, 4, 5, 6, 2, 1 };
18    int iarr2[] = { 2, 1, 4, 6, 6, 2, 1 };
19    int ret = vergleich( iarr1,(sizeof(iarr1)/sizeof(int)),
                           iarr2,(sizeof(iarr2)/sizeof(int)));
20    if( ret == DIFFSIZE ) {
21      printf("Die Arrays sind unterschiedlich lang\n");
22    }
23    else if( ret == EQUAL ) {
24      printf("Beide Arrays sind identisch\n");
25    }
26    else {
27      printf("Unterschied an Pos. %d gefunden\n", ret);
28    }
29    return EXIT_SUCCESS;
30  }
```

9. Hierzu eine Musterlösung:

```
00  // kap009/loesung003.c
01  #include <stdio.h>
02  #include <stdlib.h>
03  #include <string.h>
04  #define MAX 255

05  void ersetzen( char arr[], int n, char ch1, char ch2 ) {
06    for(int  i=0; i < n; i++ ) {
07      if( arr[i] == ch1 ) {
08        arr[i] = ch2;
09      }
```

```
  }
}

int main(void) {
  char str[MAX];
  printf("Bitte Mustertext eingeben: ");
  if( fgets( str, MAX, stdin ) == NULL ) {
    printf("Fehler bei der Eingabe\n");
    return EXIT_SUCCESS;
  }
  // 'a' gegen 'x' ersetzen
  ersetzen( str, strlen(str), 'a', 'x');
  printf("%s\n", str);
  return EXIT_SUCCESS;
}
```

B.9 Antworten und Lösungen zum Kapitel 10

1. Einfache Zeiger erkennt man am Stern * in der Deklaration zwischen dem Datentyp und dem Bezeichner. Ein Zeiger wird dazu verwendet, Adressen als Wert zu speichern, um einen bestimmten Speicherbereich im Arbeitsspeicher zu referenzieren. Damit die Zeigerarithmetik auch richtig funktioniert, muss der Datentyp des Zeigers vom selben Typ sein wie der, auf dessen Adresse er referenziert.
2. Bei falscher Initialisierung von Zeigern, beispielsweise beim Vergessen des Adressoperators, kann es schnell passieren, dass ein Zeiger auf eine ungültige Adresse im Speicher verweist. Ein Zeiger kann quasi jede beliebige Adresse im Speicher referenzieren. Die meisten Betriebssysteme haben hier zwar einen Speicherschutz und lösen einen Speicherzugriffsfehler (Segmentation fault) aus, aber dies muss nicht überall so sein, und der Standard schreibt diesbezüglich auch nichts vor.
3. Mit einer Dereferenzierung können Sie direkt auf die Werte eines Speicherobjekts zugreifen, das Sie zuvor referenziert haben. Für den direkten Zugriff wird der Indirektionsoperator * vor den Bezeichner des Zeigers gestellt.

4. Um zu vermeiden, dass ein Zeiger auf eine ungültige Adresse verweist, sollte man diesen gleich bei der Deklaration mit dem `NULL`-Zeiger initialisieren und vor jeder Verwendung eine Überprüfung auf `NULL` durchführen. Gibt die Überprüfung `NULL` zurück, hat der Zeiger noch keine gültige Adresse, und Sie können im Programm darauf reagieren. Globale Zeiger und Zeiger mit dem Schlüsselwort `static` werden automatisch mit dem `NULL`-Zeiger initialisiert.
5. Der `void`-Zeiger ist ein typenloser Zeiger. Er wird vorwiegend verwendet, wenn der Datentyp eines Zeigers, auf dessen Adresse referenziert werden soll, noch nicht feststeht. Im Gegensatz zu einem typisierten Zeiger kann ein void-Zeiger nicht dereferenziert werden.
6. Steht `const` links vom Stern, handelt es sich um konstante Daten. Steht `const` hingegen auf der rechten Seite des Sterns, handelt es sich um einen konstanten Zeiger.
7. In Zeile **(03)** wurde der Adressoperator vor `ival` vergessen. Also müsste Zeile **(03)** richtig lauten:

   ```
   03  ptr = &ival;
   ```

8. Beide Male wird der Wert 123456789 ausgegeben.
9. Die Ausgabe des Codeausschnitts lautet:

   ```
   56
   23
   3
   1
   0
   ```

   Eine kurze Erläuterung dazu: In Zeile **(03)** wird `ptr1` die Anfangsadresse auf das erste Element von `iarray` übergeben. Diese Adresse wird in Zeile **(05)** um zwei Elemente vom Typ `int` erhöht. Daher verweist der Zeiger `ptr1` in diesem Fall auf das dritte Element in `iarray`. Das ist der Wert 56, wie er in Zeile **(07)** ausgegeben wird. `ptr2` hingegen bekommt in Zeile **(04)** die Adresse des fünften Elements in `iarray` zugewiesen. In Zeile **(06)** wird die Adresse mit dem Inkrement-Operator um 1 erhöht, womit `ptr2` jetzt auf die Adresse des sechsten Elements zeigt. In diesem Fall ist der Wert 23, wie die Ausgabe der Zeile **(08)** bestätigt. Dass in Zeile **(09)** die Subtraktion von `ptr2-ptr1` den Wert 3 zurückgibt, liegt daran, dass

die Anzahl der Elemente zwischen den beiden Adressen eben gleich drei ist. Das Umwandlungszeichen `%td` wird für den Rückgabetyp `ptrdiff_t` der Subtraktion zweier Zeiger benötigt. In Zeile **(10)** gibt der Ausdruck (`ptr1 < ptr2`) wahrheitsgemäß den Wert 1 zurück, weil die Adresse von `ptr1` kleiner als die Adresse von `ptr2` ist. Zur Demonstration wurde in Zeile **(11)** noch einmal das Gleiche gemacht. Hier wurde allerdings der Indirektionsoperator verwendet. Damit haben Sie die beiden Werte von `ptr1` (=56) und `ptr2` (=23) miteinander verglichen. Aus diesem Grund wird jetzt auch 0 zurückgegeben.

10. Der Fehler liegt darin, dass die Funktion den lokalen String `buf` von Zeile **(06)** zurückgibt. Allerdings ist die Lebensdauer von lokalen Variablen in Funktionen nur zur Ausführung der Funktion gültig. Nach dem Rücksprung von der Funktion werden die Daten auf dem Stack wieder gelöscht, und `str` in der `main`-Funktion verweist auf einen undefinierten Speicherbereich. Das Problem können Sie entweder lösen, indem Sie einen statischen Puffer mit dem Schlüsselwort `static` verwenden, einen globalen String benutzen oder mit `malloc()` einen Speicher zur Laufzeit reservieren. Ein globaler String ist hier unschön, und `malloc()` kennen Sie bisher noch nicht. Daher können Sie in Zeile **(06)** das Schlüsselwort `static` vor den String stellen. Dann klappt es auch mit der Funktion.

```
06    static char buf[MAX] = " ";
```

11. Bei der ersten Version wird ein Zeiger erzeugt, der auf das Zeichen `H` in der konstanten Zeichenkette `"Hallo Welt"` zeigt. Die Zeichenkette `"Hallo Welt"` wiederum befindet sich irgendwo anders im Speicher. Mit der zweiten Version wird ein `char`-Array mit 11 Elementen erzeugt (das letzte Element ist `\0`).

12. Im Folgenden sehen Sie eine solche Musterlösung, bei der, falls eine größere Länge als vorhandene Ziffern angegeben ist, vorne mit Nullen gefüllt wird. In der Praxis wäre allerdings ein dynamisch zur Laufzeit reservierter Speicher besser geeignet als ein Puffer, der hier mit `static` gekennzeichnet wurde.

```
00  // kap010/loesung001.c
01  #include <stdio.h>
02  #include <stdlib.h>
```

```
03 #include <string.h>
04 #define MAX 255

05 char *int2string(int number, int n) {
06   static char buf[MAX];
07   int i;
08   if( n > MAX ) {
09     return ("Fehler n > MAX");
10   }
11   for( i=0; i < n; i++ ) {
12     buf[n-i-1] = (number % 10) + 48;
13     number = number / 10;
14   }
15   buf[n] = '\0';
16   return buf;
17 }

18 int main(void) {
19   int val1 = 1234, val2 = 456789;
20   char *ptr=NULL;
21   ptr = int2string(val1, 4);
22   printf("%s\n", ptr);
23   ptr = int2string(val2, 9);
24   printf("%s\n", ptr);
25   return EXIT_SUCCESS;
26 }
```

Das Programm bei der Ausführung:

```
1234
000456789
```

13. Hierzu eine Musterlösung:

```
00 // kap010/loesung002.c
01 #include <stdio.h>
02 #include <stdlib.h>

03 double Vquader(const double *a, const double *b, const double *
```

```
c) {
04    return ((*a) * (*b) * (*c));
05  }

06  int main(void) {
07    double d1 = 6.0, d2=4.1, d3=3.2;
08    printf("Quadervolumen: %lf\n", Vquader(&d1,&d2,&d3));
09    return EXIT_SUCCESS;
10  }
```

14. Eine mögliche Musterlösung zur Aufgabe:

```
00  // kap010/loesung003.c
01  #include <stdio.h>
02  #include <stdlib.h>
03  #include <string.h>
04  #define MAX 2048

05  int main(void) {
06    char buf[MAX]="";
07    const char token[] = " ,.:!?\t\n";
08    char *worte[MAX];

09    printf("Bitte den Text eingeben: ");
10    if( fgets(buf, MAX, stdin) == NULL ) {
11      printf("Fehler bei der Eingabe\n");
12      return EXIT_FAILURE;
13    }
14    char* ptr = strtok(buf, token);
15    size_t i = 0;
16    while(ptr != NULL) {
17      worte[i] = ptr;
18      ptr = strtok(NULL, token);
19      i++;
20    }
21    printf("Zerlegt in einzelne Woerter:\n");
22    for(size_t j=0; j < i; j++) {
23      printf("%s\n", worte[j]);
```

```
24    }
25    return EXIT_SUCCESS;
26  }
```

Das Programm bei der Ausführung:

```
Bitte den Text eingeben: Dieser Text wird in einzelne Woerter
zerlegt
Zerlegt in einzelne Woerter:
Dieser
Text
wird
in
einzelne
Woerter
zerlegt
```

B.10 Antworten und Lösungen zum Kapitel 11

1. Es gibt die Funktionen `malloc()`, `calloc()` und `realloc()`. Die einfachste Funktion ist `malloc()`. Mit ihr wird eine bestimmte Größe eines Speicherblocks reserviert. Die einzelnen Bytes haben dabei einen undefinierten Wert. `calloc()` hingegen initialisiert jedes Byte des reservierten Speicherblocks mit 0. `realloc()` kann neben der Möglichkeit, wie `malloc()` einfache Speicherblöcke zu reservieren, zusätzlich den Speicherblock vergrößern, verkleinern oder komplett freigeben.
2. Alle drei Funktionen geben einen typenlosen Zeiger (`void*`) mit der Anfangsadresse des ersten Bytes auf den zugeteilten Speicherblock zurück. Ein Typecasting von `void*` ist nicht nötig, weil dieses vom C-Compiler implizit durchgeführt wird.
3. Nicht mehr benötigten Speicherplatz können Sie mit der Funktion `free()`wieder freigeben.
4. Der Programmierer hat hier offensichtlich versucht, in den Zeilen **(08)** und **(09)** die Anfangsadressen der beiden dynamischen Arrays zu tauschen. Durch die Zuweisung von `iarray1=iarray2` in Zeile **(08)** ist allerdings der zuvor reservierte Speicherplatz (bzw. die Adresse) von `iarray1` für immer verloren. Es wurde ein weiteres Memory Leak erzeugt. Jetzt

verweisen beide Zeiger auf die Adresse von `iarray2`. Da hilft auch die zweite Zuweisung von `iarray2=iarray1` in Zeile **(09)** nichts mehr. Damit die Adressen tatsächlich getauscht werden können, benötigen Sie einen dritten temporären Zeiger, der sich die Adresse merkt. Im folgenden Codeausschnitt wurde dies behoben:

```
01  int *iarray1=NULL, *iarray2=NULL;
02  int *tmp=NULL;
03  iarray1 = malloc( BLK * sizeof(int) );
04  iarray2 = malloc( BLK * sizeof(int) );
...
05  for(int i=0; i<BLK; i++) {
06    iarray1[i] = i;
07    iarray2[i] = i+i;
08  }
09  tmp = iarray1;
10  iarray1 = iarray2;
11  iarray2 = tmp;
...
```

5. Hierzu eine Musterlösung:

```
00  // kap011/loesung001.c
01  #include <stdio.h>
02  #include <string.h>
03  #include <stdlib.h>
04  #define BUF 4096

05  int main(void) {
06    size_t len;
07    char *str = NULL;
08    char puffer[BUF+1];

09    printf("Text eingeben\n> ");
10    if( fgets(puffer, sizeof(puffer), stdin) == NULL ) {
11      printf("Fehler bei der Eingabe\n");
12      return EXIT_FAILURE;
13    }
14    str = malloc(strlen(puffer)+1);
```

```
15    if(NULL == str) {
16      printf("Kein virtueller RAM mehr vorhanden ... !");
17      return EXIT_FAILURE;
18    }
19    strcpy(str, puffer);
20    while(1) {
21      printf("Weiterer Text (Beenden mit ENDE)\n>");
22      if( fgets(puffer, sizeof(puffer), stdin) == NULL ) {
23        printf("Fehler bei der Eingabe\n");
24        return EXIT_FAILURE;
25      }
26      // Abbruchbedingung
27      if( strcmp(puffer,"ende\n")==0 ||
28        strcmp(puffer,"ENDE\n")==0) {
29        break;
30      }
31      // Aktuelle Länge von str zählen für realloc
32      len = strlen(str);
33      // Neuen Speicher für str anfordern
34      str = realloc(str,strlen(puffer)+len+1);
35      if(NULL == str) {
36        printf("Kein virtueller RAM mehr vorhanden ... !");
37        return EXIT_FAILURE;
38      }
39      // Hinten anhängen
40      strcat(str, puffer);
41    }
42    printf("Der gesamte Text lautet (%zd Bytes): \n", sizeof(str));
43    printf("%s", str);
44    free(str);
45    return EXIT_SUCCESS;
46  }
```

6. Hierzu eine Musterlösung. Die Funktion dazu wurde in den Zeilen **(05)** bis **(11)** erstellt. Die `main`-Funktion dient nur dazu, die Funktion zu testen.

```c
// kap011/loesung002.c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#define BLK 64

void cmp_adress(const int *adr1, const int *adr2 ) {
  if( adr1 != adr2 ) {
    printf("realloc musste umkopieren\n");
    printf("Alte Adresse: %p\n", adr1);
    printf("Neue Adresse: %p\n", adr2);
  }
}

int main(void) {
  size_t len = BLK;
  int val = 0;
  int* ivals = malloc(BLK * (sizeof(int)));
  if(NULL == ivals) {
    printf("Kein virtueller RAM mehr vorhanden ...!");
    return EXIT_FAILURE;
  }
  int* iptr = ivals;
  while(1) {
    printf("Wie viel neuer Speicher > ");
    if( scanf("%d", &val) != 1 ) {
      printf("Fehler bei der Eingabe\n");
      return EXIT_FAILURE;
    }
    // Aktuelle Länge von str zählen für realloc
    len += val;
    // Neuen Speicher für str anfordern
    ivals = realloc(ivals,len);
    if(NULL == ivals) {
      printf("Kein virtueller RAM mehr vorhanden ...!");
      return EXIT_FAILURE;
```

```
34      }
35      cmp_adress( ivals, iptr );
36      iptr = ivals;
37    }
38    return EXIT_SUCCESS;
39  }
```

Das Programm bei der Ausführung:

```
Wie viel neuer Speicher > 12345678
realloc musste umkopieren
Alte Adresse: 0000000002551048
Neue Adresse: 0000000000550290
Wie viel neuer Speicher > 2345678
realloc musste umkopieren
Alte Adresse: 0000000003122048
Neue Adresse: 0000000002551048
Wie viel neuer Speicher > 500000
realloc musste umkopieren
Alte Adresse: 0000000003f3f048
Neue Adresse: 0000000003122048
Wie viel neuer Speicher >
```

B.11 Antworten und Lösungen zum Kapitel 12

1. Mit Strukturen erstellen Sie einen eigenen zusammengesetzten Datentyp, der mehrere Komponentenvariablen mit gleichen und/oder unterschiedlichen Typen zu einem neuen Datentyp zusammenfassen kann. Bei der Deklaration wird das Schlüsselwort `struct` verwendet. Die gleichen oder unterschiedlichen Datentypen werden zwischen geschweiften Klammern zusammengefasst und als Komponenten bzw. Strukturelemente (engl. *member*) bezeichnet. Abgeschlossen wird die Deklaration mit einem Semikolon. Zugreifen kann man auf die einzelnen Komponenten mit dem Punkt-Operator (`.`). Wenn es sich bei der Strukturvariablen um einen Zeiger handelt, nutzen Sie den Pfeil-Operator (`->`).

2. Unions sind den Strukturen recht ähnlich. Der Zugriff erfolgt genauso, allerdings mit dem Unterschied, dass bei Unions die einzelnen Elemente nicht hintereinander im Speicher liegen, sondern alle mit derselben Anfangsadresse beginnen. Das bedeutet, dass immer nur ein Element in einer Union mit einem gültigen Wert belegt werden kann. War ein anderes Element bereits mit einem Wert versehen, wird dieser überschrieben. Eine Union ist also nur so groß wie das größte Element in der Union. Anstatt des Schlüsselwortes `struct` wird bei einer Union das Schlüsselwort `union` verwendet.
3. Die Strukturvariable in Zeile **(06)** erstellen Sie, indem Sie das Schlüsselwort `struct` davorstellen. Wollen Sie Zeile **(06)** trotzdem so verwenden, wie es im Beispiel angegeben ist, müssen Sie `typedef` verwenden. Hier ein Beispiel:

```
01  typedef struct artikel {
02    char schlagzeile[255];
03    int seite;
04    int ausgabe;
05  } Artikel;
...
06 Artikel artikel1;  // Jetzt klappt es auch damit
```

4. Der erste Fehler wurde in Zeile **(15)** gemacht. Dort wurde die Struktur sofort mit Werten initialisiert. Hierbei wurde die richtige Reihenfolge der einzelnen Strukturelemente nicht beachtet. Der zweite Fehler wurde in den Zeilen **(18)** bis **(20)** gemacht. Dort wurde mit einem Strukturzeiger auf Strukturelemente zugegriffen, obwohl gar kein Speicher für eine solche Struktur reserviert war. Im Beispiel wurde daher in der Zeile **(16)** ein dynamischer Speicher vom Heap reserviert. Zwar wurde hier auf eine Überprüfung verzichtet, in der Praxis sollten Sie das allerdings unterlassen. Den dritten und letzten Fehler finden Sie in den Zeilen **(21)** bis **(23)**. Dort wurde der Index-Operator (`[]`) des Struktur-Arrays an der falschen Position verwendet. Hier das Listing mit den ausgebesserten Fehlern:

```
00  // kap012/loesung001.c
01  #include <stdio.h>
02  #include <stdlib.h>
```

```
03  #include <string.h>

04  typedef struct artikel{
05    char schlagzeile[255];
06    int seite;
07    int ausgabe;
08  } Artikel;

09  void output( Artikel *a ) {
10    printf("%s\n", a->schlagzeile);
11    printf("%d\n", a->seite);
12    printf("%d\n\n", a->ausgabe);
13  }

14  int main(void) {
15    Artikel art1 = {"Die Schlagzeile schlechthin",244, 33};
16    Artikel *art2 = malloc(sizeof(Artikel));
17    Artikel artArr[2];

18    strncpy( art2->schlagzeile, "Eine Schlagzeile", 255);
19    art2->seite = 212;
20    art2->ausgabe = 43;

21    strncpy( artArr[0].schlagzeile, "Noch eine", 255);
22    artArr[0].seite = 266;
23    artArr[0].ausgabe = 67;

24    output( &art1 );
25    output( art2 );
26    output( &artArr[0] );
27    return EXIT_SUCCESS;
28  }
```

5. Sehen Sie nachfolgend eine mögliche Musterlösung. Die Funktion gibt 0 zurück, wenn beide Strukturen gleich sind; ansonsten wird 1 zurückgegeben:

```c
// kap012/loesung002.c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

typedef struct artikel{
  char schlagzeile[255];
  int seite;
  int ausgabe;
} Artikel;

int ArtikelCmp( Artikel *art1, Artikel *art2 ) {
   if( strcmp(art1->schlagzeile, art2->schlagzeile) ) {
     return 1;
   }
   else if( art1->seite != art2->seite ) {
     return 1;
   }
   else if( art1->ausgabe != art2->ausgabe ) {
     return 1;
   }
   return 0;
}

int main(void) {
  Artikel art1 = {"Die Schlagzeile schlechthin", 244, 33};
  Artikel art2 = {"Die Schlagzeile schlechthin", 244, 33};
  Artikel art3 = {"Die Schlagzeile_schlechthin", 244, 33};

  if( ArtikelCmp( &art1, &art2 ) ) {
     printf("art1 und art2 sind nicht gleich\n");
  }
  if( ArtikelCmp( &art2, &art3 ) ) {
     printf("art2 und art3 sind nicht gleich\n");
  }
  return EXIT_SUCCESS;
}
```

B.12 Antworten und Lösungen zum Kapitel 13

1. Verkettete Listen werden dynamische Datenstrukturen genannt, in denen eine unbestimmte Anzahl von gleichen Speicherobjekten gespeichert wird. Das sind gewöhnlich Strukturen, die auch als Knoten bezeichnet werden. Die einzelnen Knoten werden mithilfe von Zeigern als Komponenten vom selben Typ auf das jeweils nächste Speicherobjekt realisiert. Ein solcher Zeiger wird innerhalb eines Knotens definiert. Dadurch wird sichergestellt, dass jedes Speicherobjekt einen solchen Zeiger enthält. Damit kann eine Struktur nach der anderen ein- oder angehängt werden.
2. Verkettete Listen müssen im Gegensatz zu Arrays nicht nacheinander im Speicher abgelegt sein. Der Vorteil daran ist, dass das Löschen und Einfügen von Elementen mit einer konstanten Zeit möglich ist. Besonders das Einfügen am Anfang oder Ende einer Liste ist unschlagbar schnell. Im Gegensatz zu den Arrays muss bei einer Liste nur der Verweis auf die Adresse geändert werden. Bei den Arrays müssten hier unter Umständen ganze Elemente verschoben werden, wenn Sie keine Speicherlücken haben möchten. Allerdings sind der Aufwand und auch der Speicherverbrauch für ein Element in der Liste etwas größer. Zum einen werden die Verweise auf die einzelnen Listenelemente gespeichert, zum anderen muss die richtige Verkettung programmtechnisch sichergestellt werden. Der Nachteil von verketteten Listen ist aber auch, dass die Suche nach Daten zum Löschen oder Einfügen erheblich länger dauern kann, weil jedes einzelne Element durchlaufen (iteriert) werden muss. Auch das Einfügen des ersten und letzten Elements muss gesondert behandelt werden.
3. Bei den doppelt verketteten Listen hat jedes einzelne Element (jeder Knoten) neben einem Zeiger als Komponente auf den Nachfolger auch einen Zeiger auf den Vorgänger in der Liste. Der Vorteil ist, dass Sie eine Liste von Anfang bis Ende und umgekehrt durchlaufen können.
4. Dabei handelt es sich um einen oft gemachten Fehler in Zeile **(15)**. Dort wurde der gefundene Knoten nicht richtig »ausgehängt«. `hilfsZeiger1` zeigt hier auf die Adresse des nächsten Elements von `hilfsZeiger2`. Wenn Sie den Speicher mit `free()` freigeben, ist die verkettete Liste ab der Position `hilfsZeiger2` zerrissen, und auf die dahinterliegenden

Daten kann nicht mehr zugegriffen werden. Bildlich können Sie sich das bis zur Zeile nach `free()` wie in Abbildung B.1 vorstellen.

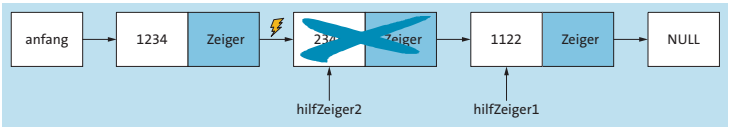

Abbildung B.1 Die verkettete Liste ist nicht mehr intakt. Der »next«-Zeiger nach dem ersten Knoten verweist auf einen nicht mehr gültigen Speicherbereich, der mit »free()« freigegeben wurde.

Hierzu der Codeausschnitt, jetzt mit fett hervorgehobener Korrektur:

```
10        ...
11        hilfZeiger1 = anfang;
12        while( hilfZeiger1->next != NULL ) {
13          hilfZeiger2 = hilfZeiger1->next;
14          if( hilfZeiger2->wert == val ) {
15            hilfZeiger1->next = hilfZeiger2->next;
16            free(hilfZeiger2);
17            break;
18          }
19          hilfZeiger1 = hilfZeiger2;
20        }
21        ...
```

5. Sehen Sie nachfolgend eine Musterlösung der neuen Funktion `einfuegenKnoten()`, die jetzt sortiert eingefügt werden kann. Zugegeben, die Aufgabe war nicht einfach, aber am Anfang und Ende einfügen sollte machbar gewesen sein. Schwieriger war da schon das Einfügen irgendwo in der Liste.

```
00  // kap013/loesung001.c
...
01  void einfuegenKnoten( KnotenPtr_t neu ) {
02    KnotenPtr_t hilfZeiger;
03    KnotenPtr_t hilfZeiger2;
04    if( anfang == NULL ) {
05      anfang = neu;
```

```
06      neu->next = NULL;
07    }
08    else {
09      hilfZeiger = anfang;
10      while(hilfZeiger != NULL &&
           (neu->wert > hilfZeiger->wert) )  {
11        hilfZeiger = hilfZeiger->next;
12      }
13      // Am Ende einfügen - Ende-Zeiger wäre sinnvoller
14      if(hilfZeiger == NULL) {
15        hilfZeiger = anfang;
16        while(hilfZeiger->next != NULL) {
17          hilfZeiger=hilfZeiger->next;
18        }
19        hilfZeiger->next = neu;
20        neu->next  = NULL;
21      }
22      // Auf doppelte Werte hin prüfen
23      else if( neu->wert == hilfZeiger->wert ) {
24        printf("Wert ist bereits vorhanden!!\n");
25      }
26      // Am Anfang einfügen
27      else if( hilfZeiger == anfang ) {
28        neu->next = hilfZeiger;
29        anfang    = neu;
30      }
31      // Irgendwo einfügen
32      else {
33        hilfZeiger2 = anfang;
34        while(hilfZeiger2->next != hilfZeiger) {
35          hilfZeiger2=hilfZeiger2->next;
36        }
37        neu->next = hilfZeiger2->next;
38        hilfZeiger2->next = neu;
39      }
40    }
41  }
```

6. Die Aufgabe sollte Sie nicht unbedingt überfordert haben. Im Grunde ähnelt diese Funktion der Funktion `knotenAuflisten()`, in der alle Elemente ausgegeben wurden. Nur müssen Sie hier die Elemente überprüfen und dann ggf. einen Hinweis geben, ob das Element in der Liste vorhanden ist oder nicht. Hierzu die Musterlösung mit der `main()`-Funktion:

```
// kap013/loesung002.c
...
void sucheKnoten( int val ) {
  KnotenPtr_t hilfZeiger = anfang;
  while( hilfZeiger != NULL ) {
    if(hilfZeiger->wert == val ) {
      printf("Wert %d gefunden\n", hilfZeiger->wert);
      return;
    }
    hilfZeiger = hilfZeiger->next;
  }
  printf("%d ist in der Liste nicht vorhanden\n", val);
}

int main(void) {
  int wahl = 0, val = 0;
  do {
    printf(" -1- Neues Element hinzufuegen\n");
    printf(" -2- Element loeschen\n");
    printf(" -3- Alle Elemente auflisten\n");
    printf(" -4- Element suchen\n");
    printf(" -5- Programmende\n");
    printf(" Ihre Auswahl : ");
    if( scanf("%d", &wahl) != 1 ) {
      printf("Fehlerhafte Auswahl\n");
      wahl = 0;
      dump_buffer(stdin);
    }
    switch( wahl ) {
      case 1 : neuerKnoten(); break;
...
```

```
        case 4 : if( anfang == NULL ) {
                   printf("Liste ist leer!\n");
                 }
                 else {
                   printf("Gesuchter Wert : ");
                   if( scanf("%d", &val) != 1 ) {
                     printf("Fehler bei der Eingabe\n");
                   }
                   else {
                     sucheKnoten( val );
                   }
                 }
                 break;
      }
    }while( wahl != 5 );
    return EXIT_SUCCESS;
}
```

B.13 Antworten und Lösungen zum Kapitel 14

1. Als Stream wird in C der Datenstrom bezeichnet. Ein neuer Stream wird gewöhnlich beim Öffnen einer Datei geöffnet und als Speicherobjekt vom Typ `FILE` dargestellt. Beim Schließen des Streams wird dieser auch wieder zerstört. Beim Start eines C-Programms stehen immer die drei Standard-Streams `stdin` (Standardeingabe), `stdout` (Standardausgabe) und `stderr` (Standardfehlerausgabe) zur Verfügung.
2. Die Funktion `fopen()` öffnet eine Datei in einem bestimmten Modus zum Lesen und/oder Schreiben. Ähnlich ist die Funktion `freopen()`, nur kann hier zusätzlich noch ein Stream umgelenkt werden. In der Praxis werden mit diesen Funktionen Standard-Streams umgelenkt, sodass beispielsweise die Standardausgabe wiederum in eine Datei umgelenkt wird. Dann gibt es noch die Funktion `tmpfile()`, mit der Sie eine temporäre Datei im Modus `"wb+"` öffnen können. Die temporäre Datei ist nur zur Laufzeit des Programms gültig und wird bei seiner Beendigung wieder gelöscht.

3. In Zeile **(05)** wurde die Datei zum Lesen mit dem Modus `"w"` geöffnet. Damit wird der komplette ursprüngliche Inhalt gelöscht. Diese Zeile sollten Sie mit dem Lesemodus (beispielsweise `"r"` oder `"r+"`) öffnen. Hier ein Beispiel:

```
05    fpr = fopen( FILENAME1, "r" );
```

4. Das Problem beim Listing ist, dass quasi 1.000 FILE-Streams auf einmal geöffnet werden. Der Standard garantiert aber nur `FOPEN_MAX` offene Streams. Viele Systeme können hier jedoch mehr Streams gleichzeitig geöffnet lassen. Daher kann das Beispiel auf dem einen System ohne Probleme laufen und auf dem anderen bereits nach `FOPEN_MAX` offenen Streams schlappmachen. Für eine einheitliche Lösung, die auf jedem System funktioniert, sollten Sie nicht mehr benötigte Streams wieder mit `fclose()` schließen. Im Beispiel sollten Sie daher in jedem Schleifendurchlauf vor der Zeile den Stream wieder schließen. Ein Array mit FILE-Zeigern wie in diesem Beispiel ist allerdings auch so überhaupt nicht nötig. Das Beispiel lässt sich mit einem FILE-Zeiger auch wie folgt realisieren:

```
// kap014/loesung001.c
#include <stdio.h>
#include <stdlib.h>

int main(void) {
  int i = 0;
  char filename[255];
  FILE *fp = NULL;
  printf("Programmstart\n");
  while( i < 1000 ) {
    sprintf( filename, "Berechnung%d.txt", i);
    fp = fopen( filename, "w" );
    if( fp != NULL ) {
      fprintf(fp, "Ergebnis der Berechnung Nr.%d", i );
      fclose(fp);
    }
    else {
      fprintf(stderr, "Ergebnis der Berechnung Nr.%d", i);
```

```
    }
    ++i;
  }
  return EXIT_SUCCESS;
}
```

5. Hier eine Musterlösung:

```
// kap014/loesung002.c
#include <stdio.h>
#include <stdlib.h>
#define CSV "datei.txt"

int main(void) {
  FILE *fp;
  char wochentag[20];
  int einnahmen, ausgaben;
  int ein_ges=0, aus_ges=0;

  fp = fopen(CSV, "r");
  if(fp == NULL) {
    fprintf(stderr, "Fehler beim Oeffnen\n");
    return EXIT_FAILURE;
  }
  printf("%20s\t%5s\t%5s\n", "Tag", "+", "-");
  while( (fscanf(fp, "%[^;];%d;%d\n",
        wochentag, &einnahmen, &ausgaben)) != EOF) {
    printf("%20s\t%5d\t%5d\n", wochentag, einnahmen, ausgaben);
    ein_ges+=einnahmen;
    aus_ges+=ausgaben;
  }
  printf("\n%20s\t%5d\t%5d", "Gesamt", ein_ges, aus_ges);
  printf("\t = %+d\n", ein_ges-aus_ges);
  return EXIT_SUCCESS;
}
```

Das Programm bei der Ausführung:

```
      Tag     +      -
   Montag  2345   2341
Dienstag   3245   1234
 Mittwoch  3342   2341
Donnerstag 2313   2341
  Freitag  2134   6298

  Gesamt   13379  14555  = -1176
```

Index

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

Z

Die Serviceseiten

Im Folgenden finden Sie Hinweise, wie Sie Kontakt zu uns aufnehmen können.

Lob und Tadel

Wir hoffen sehr, dass Ihnen dieses Buch gefallen hat. Wenn Sie zufrieden waren, empfehlen Sie das Buch bitte weiter. Wenn Sie meinen, es gebe doch etwas zu verbessern, schreiben Sie direkt an die Lektorin: *almut.poll@rheinwerk-verlag.de*. Wir freuen uns über jeden Verbesserungsvorschlag, aber über ein Lob freuen wir uns natürlich auch!

Auch auf unserer Webkatalogseite zu diesem Buch haben Sie die Möglichkeit, Ihr Feedback an uns zu senden oder Ihre Leseerfahrung per Facebook, Twitter oder E-Mail mit anderen zu teilen. Folgen Sie einfach diesem Link: *http://www.rheinwerk-verlag.de/4108*.

Zusatzmaterialien

Zusatzmaterialien (Beispielcode, Übungsmaterial, Listen usw.) finden Sie in Ihrer Online-Bibliothek sowie auf der Webkatalogseite zu diesem Buch: *http://www.rheinwerk-verlag.de/4108*. Wenn uns sinnentstellende Tippfehler oder inhaltliche Mängel bekannt werden, stellen wir Ihnen dort auch eine Liste mit Korrekturen zur Verfügung.

Technische Probleme

Im Falle von technischen Schwierigkeiten mit dem E-Book oder Ihrem E-Book-Konto beim Rheinwerk Verlag steht Ihnen gerne er Leserservice zur Verfügung: *ebooks@rheinwerk-verlag.de*.

Über uns und unser Programm

Informationen zu unserem Verlag und weitere Kontaktmöglichkeiten bieten wir Ihnen auf unserer Verlagswebsite *http://www.rheinwerk-verlag.de*. Dort können Sie sich auch umfassend und aus erster Hand über unser aktuelles Verlagsprogramm informieren und alle unsere Bücher und E-Books schnell und komfortabel bestellen. Alle Buchbestellungen sind für Sie versandkostenfrei.

Rechtliche Hinweise

In diesem Abschnitt finden Sie die ausführlichen und rechtlich verbindlichen Nutzungsbedingungen für dieses E-Book.

Copyright-Vermerk

Ihre Rechte als Nutzer

Sie sind berechtigt, dieses E-Book ausschließlich für persönliche Zwecke zu nutzen. Insbesondere sind Sie berechtigt, das E-Book für Ihren eigenen Gebrauch auszudrucken oder eine Kopie herzustellen, sofern Sie diese Kopie auf einem von Ihnen alleine und persönlich genutzten Endgerät speichern. Zu anderen oder weitergehenden Nutzungen und Verwertungen sind Sie nicht berechtigt.

So ist es insbesondere unzulässig, eine elektronische oder gedruckte Kopie an Dritte weiterzugeben. Unzulässig und nicht erlaubt ist des Weiteren, das E-Book im Internet, in Intranets oder auf andere Weise zu verbreiten oder Dritten zur Verfügung zu stellen. Eine öffentliche Wiedergabe oder sonstige Weiterveröffentlichung und jegliche den persönlichen Gebrauch übersteigende Vervielfältigung des E-Books ist ausdrücklich untersagt. Das vorstehend Gesagte gilt nicht nur für das E-Book insgesamt, sondern auch für seine Teile (z. B. Grafiken, Fotos, Tabellen, Textabschnitte).

Urheberrechtsvermerke, Markenzeichen und andere Rechtsvorbehalte dürfen aus dem E-Book nicht entfernt werden, auch nicht das digitale Wasserzeichen.

Digitales Wasserzeichen

Dieses E-Book-Exemplar ist mit einem **digitalen Wasserzeichen** versehen, einem Vermerk, der kenntlich macht, welche Person dieses Exemplar nutzen darf. Wenn Sie, lieber Leser, diese Person nicht sind, liegt ein Verstoß gegen das Urheberrecht vor, und wir bitten Sie freundlich, das E-Book nicht weiter zu nutzen und uns diesen Verstoß zu melden. Eine kurze E-Mail an *service@rheinwerk-verlag.de* reicht schon. Vielen Dank!

Markenschutz

Die in diesem Werk wiedergegebenen Gebrauchsnamen, Handelsnamen, Warenbezeichnungen usw. können auch ohne besondere Kennzeichnung Marken sein und als solche den gesetzlichen Bestimmungen unterliegen.

Haftungsausschluss

Ungeachtet der Sorgfalt, die auf die Erstellung von Text, Abbildungen und Programmen verwendet wurde, können weder Verlag noch Autor, Herausgeber oder Übersetzer für mögliche Fehler und deren Folgen eine juristische Verantwortung oder irgendeine Haftung übernehmen.

Über den Autor

Jürgen Wolf ist Softwareentwickler, Digitalfotograf und Autor aus Leidenschaft. C/C++, Perl, Linux und die Bildbearbeitung mit Photoshop Elements und GIMP sind seine Themen. Sein Traum: ein ruhiges Leben im Westen Kanadas. Und Bücher auf der Veranda schreiben. Besuchen Sie seine umfangreiche Website *www.pronix.de*.