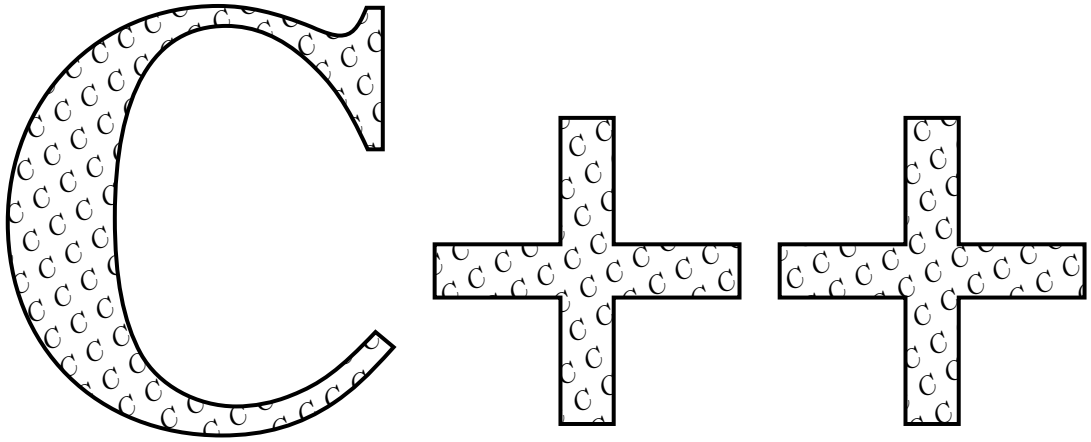
Programmierung mit C++
Teil 1

Prof. Dr.-Ing. Fritz Mehner

Fachhochschule Südwestfalen
Fachbereich Informatik und Naturwissenschaften

Version 1.3
Stand 10. Juni 2013



Inhaltsverzeichnis

Vorwort v

1. Schnellkurs *C* 1
1.1. Programmentwicklungsschritte . . . 1
1.2. Einfaches Hauptprogramm, Textausgabe, Kommentare . . . 3
1.3. Die Programmentwicklungsschritte auf der Kommandozeile . . . 6
1.4. Variablen, Ausdrücke, Zuweisungen . . . 7
1.5. Die Präprozessoranweisungen `#include` und `#define` . . . 9
1.6. Ein- und Ausgabe mit `scanf` und `printf` . . . 12
1.7. Ablaufsteuerung mit `while`, `for` und `if-else` . . . 15
1.8. Einfache Funktionen . . . 18
1.9. Felder und Zeichenketten . . . 22
1.10. Dateieingabe und Dateiausgabe . . . 27

2. Operatoren 33
2.1. Namen . . . 33
2.2. *C*-Schlüsselwörter . . . 33
2.3. Ganzzahlige Konstanten . . . 34
2.4. Inkrement- und Dekrementoperatoren . . . 34
2.5. Zuweisungsoperatoren . . . 35
2.6. Vorrang und Bindung bei Operatoren . . . 36

3. Basisdatentypen und Wertebereiche 39
3.1. Ganzzahlige Datentypen . . . 39
3.1.1. Der Datentyp `char` . . . 39
3.1.2. Der Datentyp `int` . . . 40
3.2. Reelle Datentypen . . . 41
3.3. Darstellungsbedingte Rundungsfehler . . . 43
3.4. Typumwandlung . . . 44
3.4.1. Implizite Typumwandlung . . . 44
3.4.2. Explizite Typumwandlung . . . 46

4. Ablaufsteuerung 49
4.1. Vergleichsoperatoren . . . 49
4.2. Logische Operatoren . . . 50
4.3. `if-else`-Anweisung . . . 50
4.4. `do-while`-Schleife . . . 52
4.5. `switch`-Anweisung . . . 53
4.6. `break`-Anweisung . . . 55

5. Funktionen 57
5.1. Funktionsdefinition . . . 57
5.2. Funktionsparameter mit Wertübergabe . . . 60

5.3. Prototypen . . . 62
5.4. Gültigkeitsbereich und Lebensdauer von Bezeichnern . . . 63
5.5. Speicherklassen . . . 65
5.5.1. Speicherklasse `auto` . . . 65
5.5.2. Speicherklasse `extern` . . . 65
5.5.3. Speicherklasse `static` . . . 67
5.5.4. Speicherklasse `register` . . . 69
5.5.5. Speicherklasse `volatile` . . . 69
5.6. Rekursion . . . 70

6. Felder und Zeiger **73**
6.1. Eindimensionale Felder . . . 73
6.2. Mehrdimensionale Felder . . . 73
6.3. Zeiger . . . 75
6.4. Funktionsparameter mit Adreßübergabe . . . 76
6.5. Felder und Zeiger . . . 78
6.6. Felder als Funktionsargumente . . . 79
6.7. Dynamische Speicherbelegung . . . 81
6.8. Zeiger und Zeichenketten . . . 83
6.9. Funktionen als Funktionsargumente . . . 84
6.10. Generische Programmierung in *C* . . . 86

7. Strukturen **89**
7.1. Definition und Komponentenzugriff . . . 89
7.2. Strukturen und Funktionen . . . 91

8. Bit-Operationen und Aufzählungstypen **97**
8.1. Bit-Operationen . . . 97
8.2. Aufzählungstypen . . . 101

A. Speicherbelegung eines *C*-Programmes **103**

Literaturverzeichnis **105**

Abbildungsverzeichnis **107**

Tabellenverzeichnis **109**

Programmlisten **111**

Vorwort

Die Sprache *C*

Die Sprache ***C*** wurde 1978 von den beiden Entwicklern der Sprache, Brian W. Kernighan und Dennis M. Ritchie, in ihrem Buch „The C Programming Language“ vorgestellt (das sogenannte K&R-***C***). Sie wurde in den Jahren zuvor als Implementierungssprache für das damals neue Betriebssystem ***Unix*** entwickelt. Im Jahr 1989 wurde ***C*** durch ein ANSI-Komitee[1] standardisiert. Dieser Standard wird allgemein als ANSI-***C*** oder ***C89*** bezeichnet. Er wurde 1990 wurde der Standard als ***C90*** von der ISO[2] übernommen. Im Jahre 1999 wurde eine überarbeitete Version, ***C99***, verabschiedet, die noch einige Erweiterungen brachte. Die dritte Ausgabe des Standards, kurz ***C11*** genannt (Bezeichnung ISO/IEC 9899:2011, aus den Jahr 2011), ist die derzeit gültige Version. Die ***C***-Übersetzer (oder Compiler) sind in der Regel weitestgehend mit diesem Standard verträglich.

C ist eine Systemimplementierungssprache, das heißt sie wurde dafür entworfen, Betriebssystem und Systemprogramme zu entwickeln, die außerdem möglichst einfach auf andere Hardware-Plattformen übertragbar sein sollen. Die entsprechenden Entwicklungswerkzeuge (Compiler und andere) sind fester Bestandteil aller ***Unix***/***Linux***-Systeme, da diese Systeme selbst, aber auch die meisten systemnahen Werkzeuge und viele Anwendungen, in ***C*** geschrieben sind. Die Verwendung von ***C*** ist aber keineswegs auf die Welt der Großrechner- und Server-Betriebssysteme beschränkt.

C ist die am weitesten verbreitete Programmiersprache überhaupt. Sie ist auf nahezu allen Plattformen verfügbar, vom Microcontroller bis zum Superrechner. Dementsprechend gibt es eine sehr große Anzahl von Entwicklungswerkzeugen und Bibliotheken, auf die Programmierer zurückgreifen können. Außerdem läßt sich ***C*** durch optimierende Compiler in schnell ablaufenden Maschinencode übersetzen.

Die Sprache ***C*** war und ist Ausgangspunkt für neuere, weiterentwickelte Sprachen wie zum Beispiel ***C++*** und Java. Auch Skriptsprachen wie Perl oder PHP haben viele Eigenschaften und Schreibweisen von ***C*** übernommen. Obwohl ***C*** wegen einiger Merkmale nicht ohne Kritik geblieben ist, sind die genannten Vorteile der allgemeinen Verfügbarkeit, der guten Unterstützung durch Werkzeuge und Bibliotheken, sowie die Effizienz der erzeugbaren Programme überzeugende Gründe, insbesondere für den industriellen Einsatz.

Stoffauswahl

Das vorliegende Skript enthält den Stoff für das Modul *Programmierung mit C++ 1*. Es wendet sich an die Hörer des ersten Semesters und setzt keine Programmierkenntnisse voraus. Behandelt wird, trotz des Titels, nur die Programmiersprache ***C***, die auch als Grundlage für die Einführung in ***C++*** im zweiten Semester dient.

[1] ANSI: American National Standards Institute
[2] ISO: International Organization for Standardization

Dem kundigen Leser entgeht nicht, daß der dargestellte Stoff eine Auswahl darstellt. Eine vollständige Behandlung der Programmiersprache ***C*** ist auf rund hundert Seiten nicht möglich, aber sie ist auch nicht erforderlich. Es ist das Ziel der Veranstaltung, tragfähige Grundlagen zu vermitteln und so einzuüben, daß die erworbenen Kenntnisse in späteren Anwendungen und Projekten vertieft und vervollständigt werden können.
Die gewählten Beispiele verwenden die bis dahin jeweils eingeführten Sprachmittel. Wenn weitere Kenntnisse zur Verfügung stehen, kann die eine oder andere Lösung oft eleganter formuliert werden.

Literatur

Zur Vertiefung des Stoffes und zur Begleitung des Praktikums sind in der Regel Bücher und Nachschlagewerke erwünscht. Im Literaturverzeichnis (Seite 105) sind einige Empfehlungen aufgeführt.

Zur Darstellung

Programmcode, Programmausgaben, Programm- und Dateinamen, Schlüsselwörter und Menüeinträge erscheinen in `Schreibmaschinenschrift mit fester Zeichenbreite`.

`switch`	Schlüsselwort (halbfett)
`x = 2.0-y;`	Anweisung (normal)
`/*Kommentar */`	Kommentar (kursiv)

Die Nachkommastellen reeller Zahlen werden, entsprechend der Darstellung in den meisten Programmiersprachen, durch einen *Dezimalpunkt* abgetrennt, also zum Beispiel `x = -7.11` .

1. Schnellkurs *C*

Der „Schnellkurs C“ führt einige grundlegende Konstrukte der Sprache ***C*** ein und zeigt deren Anwendung. Dazu werden kurze, vollständig ablauffähige Programm als Beispiele verwendet. Der Zweck dieses Abschnittes ist es, dem Programmieranfänger sofort das Schreiben einfachster Programme zu ermöglichen, um die Grundkonzepte zügig zu beherrschen. Die weiteren Kapitel ergänzen und erweitern diesen Stoff und führen natürlich zusätzliche Konstruktionen und Ausdrucksmöglichkeiten ein.

1.1. Programmentwicklungsschritte

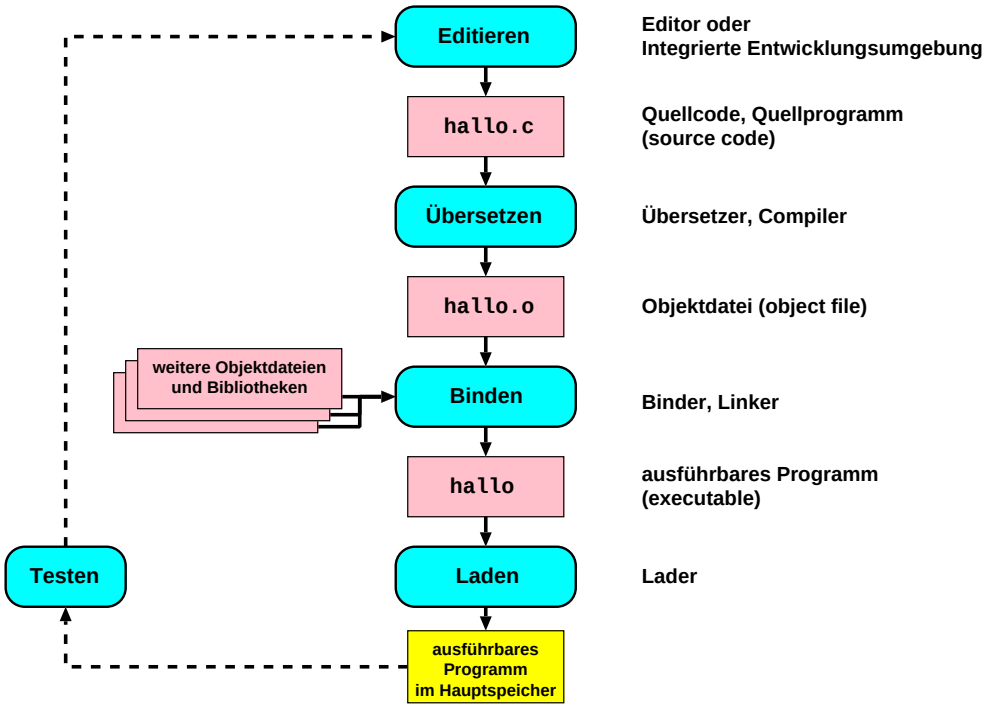


Abbildung 1.1.: Programmentwicklungsschritte

Ein Programm schreiben heißt einen Text mit Programmanweisungen einer bestimmten Programmiersprache (hier ***C***) zu erstellen. Dieser sogenannten Quelltext oder Quellcode ist nicht ausführbar. Er muß mit Hilfe eines Compilers oder Übersetzers über Zwischenstufen in ein ausführbares Programm überführt werden. Das ausführbare Programm wird getestet. Zur Behebung von Fehlern und Schwächen muß der Quelltext verändert werden. Damit beginnt der Entwicklungsprozeß wieder von vorne. Abbildung 1.1 stellt die einzelnen Schritte der Programmentwicklung in ihrer Abfolge dar. Tabelle 1.1 erläutert die Schritte etwas genauer.

Editieren	Erstellung des Programmtextes (Quellcode) mit einem Editor. Ein Programmiereditor unterstützt im Gegensatz zu einem allgemeinen Texteditor den Programmierprozeß durch viele Befehle und Hilfestellungen.
`hallo.c`	Das Ergebnis der Programmerstellung. Das Programm liegt als einfache Textdatei `hallo.c` auf der Festplatte. Ein *C*-Programm hat üblicherweise die Dateiendung `.c`
Übersetzen	Ein *C*-Compiler übersetzt die Quelldatei `hallo.c` in eine Objektdatei `hallo.o`.
`hallo.o`	Diese Datei enthält den sogenannten Objektcode. Objektcode ist nicht für den menschlichen Leser bestimmt. Er dient nur als Eingabeform für das Binden.
Binden	Der Objektcode `hallo.o` wird mit weiteren Objekten aus Programmbibliotheken zusammengebunden. Die automatisch hinzugefügten Teile enthalten unter anderem den Programmcode für die Abwicklung von Ein- und Ausgaben.
`hallo`	Das Ergebnis des Bindevorganges ist die Datei `hallo`. Sie liegt als ausführbares Programm auf der Festplatte vor.
Laden	Die ausführbare Datei `hallo` muß durch einen Aufruf an das Betriebssystem des Rechners gestartet werden (Kommandozeilenbefehl, Icon). Das Betriebssystem lädt dazu die Datei `hallo` in den Hauptspeicher und übergibt dem geladenen Programm die Kontrolle.
ausführbares Programm im Hauptspeicher	Das Programm wird durch den Prozessor ausgeführt.

Tabelle 1.1.: Programmentwicklungsschritte

1.2. Einfaches Hauptprogramm, Textausgabe, Kommentare

Die Liste 1.1 zeigt ein sehr einfaches, aber vollständiges ***C***-Programm. Es kann in der angegebenen Form übersetzt und ausgeführt werden. Die Konsolausgabe des Programmes ist in Liste 1.2 wiedergegeben.

Liste 1.1: Vollständiges ***C***-Programm (`hallo.c`) mit einfacher Textausgabe

```
1 #include <stdio.h>
2 int main ()
3 {
4   printf("\nHallo, Welt!\n");
5   return 0;
6 }
```

Liste 1.2: Ausgabe des Programmes `hallo.c`

```
1 mehner@mn4:~> ./hallo.e
2
3 Hallo, Welt!
4 mehner@mn4:~>
```

Jedes ***C***-Programm muß, unabhängig von seiner Größe, ein Hauptprogramm enthalten. Ein ***C***-Hauptprogramm hat immer die folgende Grundform:

```
int main ()
{

  return 0;
}
```

Der Name des Hauptprogrammes ist immer `main`. `main` ist ein Schlüsselwort, das heißt diese Bezeichnung darf ausschließlich für die Benennung des Hauptprogrammes verwendet werden. Hinter `main` steht ein Paar runder Klammern. Diese Klammern sind vorläufig immer leer, müssen aber trotzdem vorhanden sein. Der gesamte Text des Hauptprogrammes muß in dem nachfolgenden Paar der geschweiften Klammern `{ }` stehen.

Ein ***C***-Programm kann einen Zahlencode an das Betriebssystem oder die aufrufende Umgebung zurückgeben. Der Code könnte das System zum Beispiel darüber in Kenntnis setzen, daß das Programm ordnungsgemäß beendet wurde oder daß es durch einen internen Fehler vorzeitig beendet werden mußte. Aus diesem Grund steht vor dem Programmnamen `main` das Schlüsselwort `int` als Bezeichnung für den sogenannten Rückgabetyp. `int` ist die Bezeichnung für den Datentyp „ganze Zahl“ in ***C*** (engl. ***int****eger*). Die Festlegung `int main ()` zeigt also an, daß das Hauptprogramm nach seiner Beendigung eine ganze Zahl zurückgibt. In der Regel wird diese Zahl die Null sein. Die Null muß durch eine eigene Anweisung, die `return`-Anweisung, ausgegeben werden. Die `return`-Anweisung beendet außerdem das Hauptprogramm.

Die Zeilen 2,3,5 und 6 in Liste 1.1 können also zur Erstellung eines neuen Hauptprogrammes immer in der angegebenen Form übernommen werden.

Die Zeile 1 in Liste 1.1

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

macht beim Übersetzen und Binden die Ausgabefunktionen (hier `printf`) zugänglich. Die Bezeichnung `stdio.h` ist der Name einer Datei (Verballhornung von **st**an**d**ard **i**nput/**o**utput),

deren Speicherort dem Übersetzer bekannt ist. Im Gegensatz zu anderen Programmiersprachen (zum Beispiel FORTRAN, PASCAL) ist das grundlegende Ein-/Ausgabesystem bei *C* nicht Bestandteil der Sprache. Diese und ähnliche Zeilen können aber zunächst einfach übernommen werden. Üblicherweise stehen diese sogenannten `include`-Anweisungen am Anfang der Datei.

Textausgabe

Die Zeile 4 in Liste 1.1

```
printf("\nHallo, Welt!\n");
```

enthält die Ausgabe eines Begrüßungstextes. Diese Zeile ist, neben der `return`-Anweisung, die einzige ausführbare Anweisung des ganzen Programmes. Von außen nach innen betrachtet sehen wir zunächst einen Funktionsaufruf:

```
printf(  );
```

`print` heißt im Englischen drucken (hier auch Bildschirmausgabe), das angehängte `f` steht für formatiert, das heißt die Funktion `printf` stellt eine formatierte Ausgabe zur Verfügung.

Der auszugebende Text wird als sogenanntes Aufrufargument an die Funktion übergeben. Aufrufargumente stehen grundsätzlich innerhalb der runden Argumentklammern. Hier ist das Argument ein Text oder besser eine Zeichenkette. Zeichenketten werden in *C* grundsätzlich in doppelte Anführungszeichen gesetzt:

```
"\nHallo, Welt!\n"
```

Eine Zeichenkette wird unabhängig von ihrer Länge als ein einzelnes Argument betrachtet. Diese Zeichenkette enthält, neben zwei Wörtern und zwei Satzzeichen, am Anfang und am Ende ein sogenanntes Steuerzeichen, nämlich den Zeilenvorschub `\n`.

Diese Zeichenkombination wird nicht unmittelbar auf dem Bildschirm dargestellt sondern wirkt als Zeilenvorschub: die Schreibmarke wird auf dem Bildschirm an den Beginn der nächsten Zeile gesetzt. Vor der Ausgabe des Textes

```
Hallo, Welt!
```

wird also eine neue Zeile begonnen, danach wird ebenfalls ein Zeilenvorschub ausgegeben. Damit steht der eigentliche Text allein in einer eigenen Zeile (siehe Liste 1.2, Zeile 3).

Das Zeichen \ , der Rückstrich (engl. *backslash*), ist ein sogenanntes Fluchtsymbol. Ein Fluchtsymbol zeigt an, daß das unmittelbar nachfolgende Zeichen nicht in seiner angegebenen Form dargestellt wird, sondern in einer festgelegten Art und Weise als Steuerzeichen aufzufassen ist. Tatsächlich ist die Kombination `\n` die Ersatzdarstellung für das Steuerzeichen „Zeilenvorschub“ des ASCII-Codes (Codierung: hexadezimal 10). Weitere gebräuchliche Steuerzeichen sind in Tabelle 1.4 angegeben.

Kommentare

In einem *C*-Programm können Leerzeichen, Tabulatoren und Leerzeilen freizügig zur Programmgestaltung eingesetzt werden. Man verwendet diese Freiheit üblicherweise, um einem Programm ein gut leserliches Erscheinungsbild zu geben.

Ein Programmtext kann (muß !) außerdem mit Kommentaren versehen werden. Kommentare werden vom Übersetzer überlesen und haben keinen Einfluß auf das ausführbare Programm.

Sie dienen nur dem menschlichen Leser zur Dokumentation und sollten dazu auch ausgiebig und sinnvoll genutzt werden.

Liste 1.3: *C*-Programm mit verschiedenen Kommentarformen

```
1  /*
2   * ===========================================================================
3   *
4   *       Filename:  hallo-com.c
5   *
6   *    Description:  Vollständiges C-Programm mit einfacher Textausgabe
7   *
8   *        Version:  1.0
9   *        Created:  22.10.2006 17:43:31 CEST
10  *       Revision:  none
11  *       Compiler:  gcc
12  *
13  *         Author:  Dr.-Ing. Fritz Mehner (Mn), mehner@fh-swf.de
14  *        Company:  Fachhochschule Südwestfalen, Iserlohn
15  * ===========================================================================
16  */
17
18 #include <stdio.h>
19
20 /* Hauptprogramm (dies ist ein klassischer C-Kommentar) */
21
22 int main ()
23 {
24   printf("\nHallo, Welt!\n");     // Zeilenendkommentar
25   return 0;
26 }
27
28 /*
29  * Hier ist das Hauptprogramm zu Ende
30  */
```

Der klassische *C*-Kommentar schließt den Kommentartext in die Zeichenfolge `/*  */` ein (Liste 1.3, Zeile 20). Diese Kommentare können sich bei Bedarf über mehrere Zeilen erstrecken (Liste 1.3, Zeilen 1-16, 28-30).

Die Programmiersprache *C++* hat den Zeilenendkommentar eingeführt, der von der Zeichenfolge `//` eingeleitet wird. Der nachfolgende Text wird bis zum Zeilenende als Kommentar aufgefaßt (Liste 1.3, Zeile 24). Diese Kommentierungsform kann nicht mehrzeilig sein. Die meisten *C*-Compiler lassen beide Kommentarformen zu.

Es ist dringend anzuraten, jede Datei mit einem Kopfkommentar, dem sogenannten Dateiprolog, zu versehen. Der Dateiprolog enthält alle wichtigen Angaben zur Datei, wie etwa Name, Zweck, Projekt, Compiler, Erstellungsdatum, Änderungsdatum, Autor und so weiter (Liste 1.3, Zeilen 1-16).

Ein entsprechender Kommentarblock kann zum Beispiel aus einer Vorlagendatei kopiert werden oder von einem erweiterbaren Editor in die Programmdatei eingesetzt werden.

1.3. Die Programmentwicklungsschritte auf der Kommandozeile

Compiler, Binder, Präprozessoren und so weiter sind in der Regel eigenständige Programme, die von der Kommandozeile aus aufrufbar sind. Bei der Verwendung einer graphischen Entwicklungsumgebung (eine sogenannte IDE, **I**ntegrated **D**evelopment **E**nvironment) wird das nicht sichtbar, weil die erforderlichen Teilschritte (Übersetzen, Binden, Starten und so weiter) durch Bedienelemente oder Tastenkürzel ausgeführt werden können. Der Ablauf der Dienstprogramme findet, für den Benutzer nicht direkt sichtbar, unter der Oberfläche statt. Diese Dienstprogramme werden von der Entwicklungsumgebung gestartet und deren Ablauf überwacht. Nur die Rückmeldungen, wie etwa Erfolgs- oder Fehlermeldungen, werden in der Entwicklungsumgebung angezeigt. In derselben Weise sind diese Dienstprogramme aus Programmiereditoren heraus nutzbar. Genau diese Arbeitsweise ist in der Regel erwünscht, um eine schnelle und bequeme Programmentwicklung zu ermöglichen.
Um sich jedoch über den grundsätzlichen Ablauf der Programmentwicklungsschritte aus Abschnitt 1.1 klar zu werden, können diese Schritte auch einzeln und nacheinander auf der Kommandozeile ausgeführt werden. Auf einem ***Unix***-/***Linux***-System wird hierzu ein sogenanntes Shell-Fenster geöffnet. In diesem Fenster erscheint dann die Eingabeaufforderung des Kommandointerpreters (der sogenannte shell prompt). Nun können Befehle eingegeben werden.

Wenn das Programm in Liste 1.1 in der Datei `hallo.c` abgespeichert wurde, dann kann zunächst geprüft werden, ob diese Datei vorhanden ist:

```
mehner@mn4:~> ls -l *.c
-rw-r--r--  1 mehner users   629 2005-09-05 16:50 hallo.c
-rw-r--r--  1 mehner users 49678 2005-09-06 15:15 test1.c
mehner@mn4:~>
```

Der erste Teil der ersten Zeile ist der shell prompt: `mehner@mn4:~>` . Er zeigt hier den Benutzernamen und den Rechnernamen. Der nachfolgende Befehl `ls -l *.c` erzeugt eine Liste alle Dateien mit der Erweiterung `.c`. Die Option `-l` sorgt dafür, daß zu den Dateinamen weitere Angaben, wie zum Beispiel Datum der letzten Änderung, ausgegeben werden. Nun kann die Datei `hallo.c` übersetzt werden:

```
mehner@mn4:~> gcc -c hallo.c
mehner@mn4:~> ls -l hallo*
-rw-r--r--  1 mehner users 629 2005-09-05 16:50 hallo.c
-rw-r--r--  1 mehner users 896 2005-09-06 16:35 hallo.o
```

Der Befehl `gcc -c hallo.c` ruft den ***C***-Compiler `gcc` auf und veranlaßt die Übersetzung (Schalter `-c`) der Datei `hallo.c`. Die nachfolgende Ausgabe der Dateiliste zeigt, daß nun eine Objektdatei `hallo.o` vorhanden ist. Der Compiler dient auch als Aufrufschnittstelle für den Binder:

```
mehner@mn4:~> gcc -o hallo hallo.o
mehner@mn4:~> ls -l hallo*
-rwxr-xr-x  1 mehner users 7009 2005-09-06 16:43 hallo
-rw-r--r--  1 mehner users  629 2005-09-05 16:50 hallo.c
-rw-r--r--  1 mehner users  896 2005-09-06 16:35 hallo.o
```

Der Schalter `-o hallo` erzwingt den Dateinamen `hallo` für die zu erstellende, ausführbare Datei. Danach folgt die Auflistung der zu bindenden Objekte, die hier nur aus `hallo.o` besteht. Die nachfolgend erzeugte Dateiliste zeigt, daß eine neue, ausführbare Datei vorhanden ist, die jetzt ausgeführt werden kann:

```
mehner@mn4:~> ./hallo

Hello, World!
```

1.4. Variablen, Ausdrücke, Zuweisungen

Zur Umrechnung einer Fahrenheit-Temperatur in eine Celsius-Temperatur ist die in Gleichung 1.1 angegebene Umrechnungsvorschrift zu verwenden.

$$^\circ C = \frac{5}{9}\,(\,^\circ F - 32\,) \tag{1.1}$$

Zur Umrechnung eines gegebenen Fahrenheitwertes in einem Programm, müssen Programmgrößen verwendet werden, die in der Lage sind, Zahlenwerte zu speichern. Derartige Größen heißen wie in der Mathematik Variablen und besitzen eine Datentyp.

Liste 1.4: Einzelne Umrechnung von $^\circ F$ nach $^\circ C$

```
1  #include  <stdio.h>
2
3  int main ( )
4  {
5    double    fahrenheit, celsius;                /* Vereinbarungen */
6
7    printf("\nUmwandlung von Grad Fahrenheit in Grad Celsius\n\n");
8
9    fahrenheit  = 100.0;
10   celsius     = (5.0/9.0)*(fahrenheit - 32.0);
11   printf( "%lf Grad Fahrenheit entspr. %lf Grad Celsius\n\n", fahrenheit, celsius );
12
13   return 0;
14 }
```

In der Liste 1.4 ist ein Programm angegeben, welches die Umrechnung für den Fahrenheitwert 100° löst. Liste 1.5 zeigt den Aufruf des Programmes von der Kommandozeile und die Programmausgabe.

Liste 1.5: Ausgabe des Programmes in Liste 1.4

```
1  mehner@mn4:~> ./fahrenheit0
2
3  Umwandlung von Grad Fahrenheit in Grad Celsius
4
5  100.000000 Grad Fahrenheit entspr. 37.777778 Grad Celsius
6
7  mehner@mn4:~>
```

Für die beiden Temperaturen sind im Umrechnungsprogramm zwei Variablen einzurichten, die reelle Zahlenwerte aufnehmen können. Diese Vereinbarung oder Deklaration geschieht in Zeile 5. Die Namen der beiden Größen lauten `fahrenheit` und `celsius`. Variablennamen können in gewissen Grenzen frei gewählt werden. Groß- und Kleinschreibung wird unterschieden.

Alle Variablen eines Programmes müssen vor der Verwendung vereinbart werden: In ***C*** besteht **Vereinbarungszwang**.

Für Variablen ist außerdem immer ein **Datentyp** anzugeben. Der Datentyp lautet hier `double`. Der Typ `double` bezeichnet reelle Größen mit doppelter interner Darstellungsgenauigkeit, im

Gegensatz zum ebenfalls reellen Datentyp `float` mit einfacher interner Darstellungsgenauigkeit[1].

Weil bei einer Umrechnung einer Ganzzahl mit dem Faktor $\frac{5}{9}$ auch dann Nachkommastellen auftreten können wenn der Fahrenheitwert ganzzahlig ist, ist für den zu errechnenden Celsiuswert eine reelle Variable `celsius` zu wählen. Da bei zukünftigen Berechnungen der Fahrenheitwert ebenfalls reell sein könnte, wird die Variable `fahrenheit` ebenfalls als reelle Größe vereinbart.

`int`	ganze Zahl (engl. ***int****eger*)
`float`	reelle Zahl, einfach genau
`double`	reelle Zahl, doppelt genau
`char`	einzelnes Zeichen (engl. ***char****acter*)

Tabelle 1.2.: Basisdatentypen in *C*

In der Tabelle 1.2 sind die vier zahlenwertigen Basisdatentypen von *C* angegeben, zu denen auch der Typ `char` zählt.

Die Zeile 9 in Liste 1.4 enthält eine Wertzuweisung:

```
fahrenheit  = 100;
```

Die Variable `fahrenheit`, die bisher keinen Wert hatte, erhält nun den Wert 100. In Zeile 10 findet nun die eigentliche Umrechnung statt:

```
celsius     = (5.0/9.0)*(fahrenheit - 32.0);
```

Die Rechenoperatoren sind vom Taschenrechner bekannt. Die Reihenfolge der Schritte wird durch die Klammerung erzwungen.

Wichtig ist hier die Darstellung des Faktors $\frac{5}{9}$ in der Form `(5.0/9.0)`. Zunächst stellt man fest, daß die jeweilige Nachkommastelle 0 des Zählers und des Nenners durch einen **Dezimalpunkt** abgetrennt ist. Beide Zahlen werden hierdurch zu reellen Konstanten. Die Division liefert deshalb auch ein reelles Ergebnis, nämlich 0.5555 . . . .

Eine Darstellung in der Form `(5/9)` würde bei der Division den Wert 0 ergeben. In diesem Fall würde der Übersetzer die beiden Größen `5` und `9` als ganzzahlige Konstanten betrachten. Das Ergebnis einer ganzzahligen Division ist immer selbst ganzzahlig. Das Ergebnis wäre 0, da 9 in 5 nullmal ganz enthalten ist!

`5/9`	ganzzahlige Division	Ergebnis 0
`5.0/9.0`	reelle Division	Ergebnis 0.5555 . . .

Die unbeabsichtigte Verwendung einer ganzzahligen Division ist gelegentlich Ursache für schwer auffindbare Fehler! Die besten Gegenmaßnahmen sind Überlegung und Sorgfalt bei der Programmerstellung.

Die Zeile 11 in Liste 1.4 enthält die Ausgabe des Berechnungsergebnisses:

[1] Die Bezeichnungen erklären sich aus der Geschichte von *C*

```
printf( "%lf Grad Fahrenheit entspr. %lf Grad Celsius\n\n",
         fahrenheit, celsius );
```

Die printf-Anweisung ist umfangreicher als im ersten Programm. Sie hat folgenden allgemeinen Aufbau:

printf(***Formatbeschreibung*** , ***Argumentliste***);

Das erste Argument ist immer eine Zeichenkette. Die Argumentliste ist eine Folge von auszugebenden Größen, die durch Kommata getrennt sind. Die Argumentliste besteht hier aus den beiden Variablen fahrenheit und celsius.

In der Formatbeschreibung sind neben einfachem Text und dem Steuerzeichen \n Formatierungsanweisungen %lf für die beiden auszugebenden reellen Größen enthalten:

```
"%lf Grad Fahrenheit entspr. %lf Grad Celsius\n\n"
```

Diese beiden Teilformate %lf bezeichnen die Stellen im Text, an denen die in der Argumentliste aufgeführten Größen auszugeben sind. Die Zuordnung geschieht der Reihe nach von links nach rechts. Da in diesem Beispiel noch keine Angaben über die zu verwendende Stellenzahl vorhanden ist, wird eine Standarddarstellung mit sechs Nachkommastellen gewählt (siehe Liste 1.5).

1.5. Die Präprozessoranweisungen #include und #define

Ein *C*-Compiler ruft vor der eigentlichen Übersetzung des Quelltextes den *C*-Präprozessor auf. Die Zeilen des Quelltextes, die mit dem Zeichen # beginnen, sind Anweisungen an diesen Präprozessor. Mit Hilfe von Präprozessoranweisungen können Dateien in den Quelltext einkopiert werden (sogenannte include-Dateien), Texte durch andere ersetzt werden und Quellcodezeilen von der Übersetzung ausgeschlossen werden (bedingte Übersetzung).

#include-Anweisungen

Zu jedem vollständig und richtig eingerichteten *C*-Compiler gehören Systemverzeichnisse, die sogenannten header-Dateien (kurz h-Dateien) enthalten. Die einzelnen h-Dateien wiederum enthalten Funktionsdeklarationen, Makros, Typdefinitionen und Konstanten. Da im Gegensatz zu anderen Programmiersprachen übliche Bibliotheksfunktionen (zum Beispiel Ein-/Ausgabe, mathematischer Funktionen, Zeichenkettenbearbeitung, Zeit- und Datumsfunktionen) bei *C* nicht Bestandteil der Sprache sind, müssen die dafür vorhandenen und standardisierten Bibliotheken wie Fremdbibliotheken behandelt werden. Um Funktionen der Ein-/Ausgabe zu verwenden (zum Beispiel printf) muß deshalb am Anfang der Datei die include-Anweisung

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

stehen. Die Datei stdio.h enthält die für die Übersetzung und Einbindung der Ein-/Ausgabefunktionen notwendigen Angaben. Entsprechend ist die Anweisung

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

für die Nutzung mathematischer Funktionen (zum Beispiel sin(), cos(), tan()) erforderlich. Die *C*-Standardbibliothek enthält 29 h-Dateien (siehe Tabelle 1.3). Zu den jeweiligen *C*-Funktionen wird deshalb in Lehrbüchern und Programmieranleitungen meist die zugehörige

`include`-Datei angegeben. Im Zweifelsfall kann auf ***Unix***-/***Linux***-Systemen die Systemdokumentation abgefragt werden (zum Beispiel von der Kommandozeile: `man printf`).

Die Kenntnis der genauen Lage der h-Dateien (Verzeichnispfad) ist für den Anwender nicht erforderlich.

`assert.h`	Fehlerdiagnostik
`complex.h`	Arithmetik mit komplexen Zahlen
`ctype.h`	Klassifizierung und Umwandlung von Zeichen
`errno.h`	Fehlerdiagnostik
`fenv.h`	Zugang zur Umgebung der Gleitkommazahlen
`float.h`	Grenzwerte und Eigenschaften der Gleitkommazahlen
`inttypes.h`	Funktionen zum Umgang mit Ganzzahlen
`iso646.h`	Makros (Namen) für logische Operatoren
`limits.h`	Grenzwerte für ganzzahlige Typen
`locale.h`	nationale Anpassungen und Vereinbarungen
`math.h`	mathematische Funktionen
`setjmp.h`	Globale Sprünge
`signal.h`	Signale für die Ausnahmebehandlung
`stdalign.h`	Makros für die Speicherausrichtung von Größen
`stdarg.h`	Variable Parameterlisten
`stdatomic.h`	Makros und Datentypen für atomare Operationen
`stdbool.h`	Datentyp `bool`, Werte `true`, `false`
`stddef.h`	allgemeine Definitionen
`stdint.h`	Ganzzahlige Typen mit festgelegter Breite
`stdio.h`	Ein- und Ausgabe
`stdlib.h`	Definitionen der Standardbibliothek
`stdnoreturn.h`	Definition des Makros `noreturn`
`string.h`	Zeichenkettenbearbeitung
`tgmath.h`	generische Makros
`threads.h`	Unterstützung von threads
`time.h`	Datum- und Zeitbehandlung
`uchar.h`	Behandlung von Unicode-zeichen
`wchar.h`	Behandlung von wide-character
`wctype.h`	Klassifizierung von wide-character

Tabelle 1.3.: ***C***-Standardbibliothek, Header-Dateien (***C 11***)

`#define`-Anweisungen

Eine Besonderheit der ***C***-Sprachfamilie sind die sogenannten `define`-Anweisungen. Diese erlauben einzelne Zeichenketten, die im nachfolgenden Programmtext vorkommen, durch andere Zeichenketten zu ersetzen. Die allgemeine Form einer `define`-Anweisung lautet:

`#define` ***Originaltext Ersetzungstext***

Vor der eigentlichen Übersetzung sucht der Präprozessor alle in `define`-Anweisungen vorkommenden Originaltexte und ersetzt diese durch die entsprechenden Ersetzungstexte. Durch die Anweisung

```
#define  N   100
```

werden alle im Programmtext einzeln stehenden Buchstaben N durch die Zahl 100 ersetzt. Aus der Quelltextzeile

```
while ( i < N )
```

wird nach der Behandlung durch den Präprozessor die Zeile

```
while ( i < 100 )
```

Der Präprozessor erzeugt eine Zwischenstufe des Quelltextes, die dann vom Compiler übersetzt wird. Die vom Programmierer geschriebene Quelldatei auf der Festplatte bleibt natürlich unverändert. Bei der Ersetzung gibt es eine Ausnahme. Wenn der Originaltext einer define-Anweisung innerhalb einer Zeichenkette (eingerahmt durch doppelte Anführungszeichen) steht, wird er nicht ersetzt. Die Anweisung

```
printf("Dieses Programm verwendet die Größe N");
```

gibt den Text unverändert aus. Folgende Beispiele zeigen weitere Verwendungsmöglichkeiten

```
#define    PIHALB   (2.0*atan(1.0))
#define    PI       (2.0*PIHALB)
#define    PI2      (2.0*PI)
#define    CR       printf("\n")
```

Die Verwendung einer bereits definierten define-Größe ist möglich. So wird etwa PIHALB im Programmtext durch (2.0*atan(1.0)) ersetzt. Da es sich um eine reine Textersetzung handelt, prüft der Präprozessor nicht nach, ob die Zahlendarstellung fehlerfrei ist.

Wie auch in den Beispielen zu sehen, werden die Originaltexte in define-Anweisungen traditionell groß geschrieben, um sie im Schriftbild leichter erkennen zu können. Der Präprozessor bietet viele weitere interessante Möglichkeiten. Es ist jedoch ratsam, insbesondere die Textersetzung auf das notwendige Mindestmaß zu beschränken. Üblich ist die Verwendung von define-Anweisungen zur Festlegung von Feldgrößen oder Problemgrößen (zum Beispiel maximaler Polynomgrade und ähnliches), die in einer Datei oder einem Projekt oft vorkommen. Die define-Anweisung bietet hier eine Möglichkeit, eine später durchzuführende Änderung an genau einer Stelle durchzuführen. Nach einer Neuübersetzung ist die Änderung dann in allen Programmteilen wirksam.

Liste 1.6: Verwendung von define-Makros

```
1  #include  <math.h>
2  #include  <stdio.h>
3
4  #define    PIHALB   (2.0*atan(1.0))          /* Pi plattformunabhängig definieren */
5  #define    PI       (2.0*PIHALB)
6  #define    PI2      (2.0*PI)
7
8  int main ( void )
9  {
10   double  faktor1  = (1.0/9.0)*PI2;
11   double  faktor2  = 1.0/PI;
12   double  faktor3  = 4.0*PI2;
13   printf ( "(1) %8.4lf\n(2) %8.4lf\n(3) %8.4lf\n", faktor1, faktor2, faktor3 );
14   return 0;
15 }   /* ----------  end of function main  ---------- */
```

Liste 1.7: Programm in Liste 1.6 nach dem Präprozessorlauf (ohne die Teile aus `math.h`)

```
1   ...
2
3  int main ( void )
4  {
5    double faktor1 = (1.0/9.0)*(2.0*(2.0*(2.0*atan(1.0))));
6    double faktor2 = 1.0/(2.0*(2.0*atan(1.0)));
7    double faktor3 = 4.0*(2.0*(2.0*(2.0*atan(1.0))));
8
9    return 0;
10 }
```

In Liste 1.6 ist ein kurzes Programm angegeben, in welchem drei Makros zur Zuweisung von Anfangswerten verwendet werden. Die bei der Übersetzung erzeugte Zwischenform des Programmes kann mit Hilfe einer Aufrufoption des Übersetzers in eine Datei umgeleitet werden. Das Ende dieser Datei ist in Liste 1.7 dargestellt. Am Beginn wurde die Datei `math.h` einkopiert. Dieser Teil ist hier nicht dargestellt. Die `define`-Anweisungen sind verschwunden, dafür sind die entsprechenden Ersetzungen vorgenommen worden.

1.6. Ein- und Ausgabe mit `scanf` und `printf`

Die Verwendung der Funktionen `scanf` und `printf` stellt eine leicht zu erlernende Möglichkeit dar, die Ein-/Ausgabe von Kommandozeilenprogrammen zu gestalten. Das ist besonders für den Programmieranfänger günstig, da die Einarbeitung in die Handhabung umfangreicher graphischer Oberflächen zugunsten der Erlernung der grundlegenden Sprachelemente zunächst zurückgestellt werden kann.

Später muß dann die Programmierung graphischer Oberflächen hinzukommen, da die meisten Anwendungen graphikorientiert sind. Aus diesem Grund wird die Darstellung und Verwendung der kommandozeilenorientierten Ein-/Ausgabe in diesem Kurs auf das Notwendige beschränkt.

Die Verwendung von `printf`

Die `printf`-Anweisung hat folgenden allgemeinen Aufbau:

`printf(` ***Formatbeschreibung*** `,` ***Argumentliste*** `);`

Das erste Argument ist immer eine Zeichenkette. Die Argumentliste ist eine Folge von auszugebenden Größen, die durch Kommata getrennt sind. Die Formatbeschreibung kann Steuerzeichen des ASCII-Codes zur Gestaltung der Ausgabe enthalten. Die gebräuchlichsten sind in Tabelle 1.4 wiedergegeben.

Da ein einzelner Rückstrich ein Steuerzeichen einleitet, benötigt er selbst eine Ersatzdarstellung, wenn er in einem Text erscheinen soll: er muß zur Darstellung verdoppelt werden.

Bei der Ausgabe von Zahlen und Zeichenketten ist meistens die Festlegung der Anzahl der Vor- und Nachkommastellen, die Art der Vorzeichendarstellung, die Breite von Texten in Tabellen und so weiter erforderlich. Aus diesem Grund wird die Position und die Gestaltung der in einer `printf`-Anweisung auszugebenden Größen durch Teilformate festgelegt, die an der entsprechenden Stelle im 1. Argument, der Formatbeschreibung, stehen. Wie in Abschnitt 1.4 zu sehen war, besteht ein Teilformat im einfachsten Fall aus einem Prozentzeichen, gefolgt

`\a`	Signalton (**a**lert)
`\b`	Rückschritt (**b**ackspace)
`\f`	Seitenvorschub (**f**ormfeed)
`\n`	Zeilenvorschub (**n**ewline)
`\r`	Wagenrücklauf (carriage **r**eturn)
`\t`	Tabulator, waagerecht (horizontal **t**abulator)
`\0`	binäre Null, internes Abschlußzeichen von Zeichenketten
`\\`	Rückstrich

Tabelle 1.4.: Ersatzdarstellung einiger Steuerzeichen

von einer Typkennzeichnung (zum Beispiel `%f` für eine `double`-Größe). Das Prozentzeichen ist ein Fluchtsymbol, das die Bewertung der nachfolgenden Zeichen als Teilformat erzwingt. Die Teilformate haben folgenden allgemeinen Aufbau:

`%`[*Flags*][*Breite*][`.`*Genauigkeit*]*Typ*

Die Bestandteile in eckigen Klammern sind wahlfrei, das heißt sie können fehlen.

Breite bezeichnet die auszugebende *Gesamtzeichenzahl* bei Zahlen und Texten. Bei Zahlen zählen zur Gesamtzeichenzahl auch die zur Darstellung des Vorzeichens, des Dezimalpunktes und des Exponenten erforderlichen Zeichen.

Genauigkeit bezeichnet die Anzahl der *Nachkommastellen* bei reellen Zahlen.

Eine Auswahl von Flags und Typkennzeichnungen ist in Tabelle 1.5 wiedergegeben. Weitere Möglichkeiten können der Dokumentation der Funktion `printf` entnommen werden.

Der Programmausschnitt

```
double x;
int    i;
  x = 7.0/9.0;
  i = 999;
  printf("x=%6.2f,  i=%6d\n", x, i );
```

erzeugt die Ausgabezeile

```
x=␣␣0.78,␣␣i=␣␣␣999
```

Die Leerzeichen sind zur besseren Überprüfbarkeit mit dem Zeichen ␣ markiert. Man erkennt, daß die Darstellung des Wertes von `x` insgesamt 6 Stellen verwendet. Die führenden Leerzeichen und der Dezimalpunkt zählen zur Gesamtzeichenzahl der Darstellung. Die Darstellung des Wertes von `i` verwendet ebenfalls 6 Zeichen. Da der Wert dreistellig ist, werden bei rechtsbündiger Darstellung 3 führenden Leerzeichen eingefügt.

Die Leerzeichen nach dem ersten Komma in der `printf`-Anweisung gehören zum Ausgabetext und werden natürlich einfach übernommen.

Flag	Bedeutung
-	linksbündige Ausgabe
+	numerische Ausgabe mit Vorzeichen; bei positiven Werten wird ein Pluszeichen vorangestellt
␣	(Leerzeichen); bei positiven Werten wird ein Leerzeichen vorangestellt

Typ	Parameter	Datentyp
d	ganze Zahl	`signed int` (dezimal)
o	ganze Zahl	`unsigned int` (oktal)
u	ganze Zahl	`unsigned int` (dezimal)
x	ganze Zahl	`unsigned int` (hexadezimal)
f	reelle Zahl	`double` (Darstellung ohne Exponent)
e	reelle Zahl	`double` (Darstellung mit Exponent)
c	Zeichen	`char` (einzelnes Zeichen)

Tabelle 1.5.: Flags und Typangaben in `printf`-Formaten (Auswahl)

Die Verwendung von `scanf`

Die Funktion `scanf` dient zum Einlesen von Zahlen und Texten von der Kommandozeile. Die `scanf`-Anweisung hat folgenden allgemeinen Aufbau:

`scanf(` ***Formatbeschreibung*** `,` ***Argumentliste*** `);`

Mit der Anweisung

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

kann eine ganze Zahl in die `int`-Variable `laenge` eingelesen werden. Die Angabe des Variablennamen reicht bei der `scanf`-Funktion nicht aus. Vor jedem Argument der Argumentliste muß zwingend der sogenannte Adreßoperator `&` stehen[2].

Für die Formatbeschreibung gilt grundsätzlich alles was bei der `printf`-Funktion gesagt wurde. So ist zum Beispiel zur Eingabe einer reellen Zahl die genaue Angabe der Gesamtzeichenzahl und der Genauigkeit möglich. Diese Vorgaben müßten dann bei der Eingabe auch eingehalten werden. Da das in der Regel lästig ist, läßt man bei Dialogeingaben diese Angaben weg und beschränkt sich auf die Angabe des Datentyps. Eine `double`-Zahl wird also einfach mit dem Teilformat `%f` eingelesen.

Mehrfacheingaben sind möglich. So werden durch die nachfolgenden Zeilen in einer Anweisung zwei `int`-Werte eingelesen, die bei der Eingabe einfach durch ein Leerzeichen getrennt werden können.

```
printf("Eingabe von i und j : ")
scanf( "%d%d", &i, &j )
```

[2] In *C* werden Funktionsparameter im Ersatzfall als Wert übergeben. Die Adreßübergabe muß durch den Adreßoperator erzwungen werden.

1.7. Ablaufsteuerung mit `while`, `for` und `if-else`

In Abschnitt 1.4 wurde eine einzelne Temperaturumrechnung von Celsius nach Fahrenheit durchgeführt. Nun soll eine Umrechnungstabelle erstellt werden. Die Anforderungen an die Tabelle lauten:

- zweispaltige Tabelle
- linke Spalte: Fahrenheitwerte von 0 bis 300 Grad, Schrittweite 20 Grad
- rechte Spalte: entsprechende Celsiuswerte

Abbildung 1.2 zeigt einen Programmablaufplan, der die Schritte beschreibt, die zur Erstellung der Tabelle notwendig sind. Die Idee besteht darin, die Tabelle zeilenweise zu berechnen und die gerade berechnete Zeile (Fahrenheitwert mit zugehörigem Celsiuswert) auszugeben. Die Wiederholung wird solange durchgeführt, bis die Höchsttemperatur von 300 $[°F]$ erreicht ist. Der Programmablaufplan ist bereits so angelegt, daß er einfach in vorhandene *C*-Konstrukte umgesetzt werden kann.

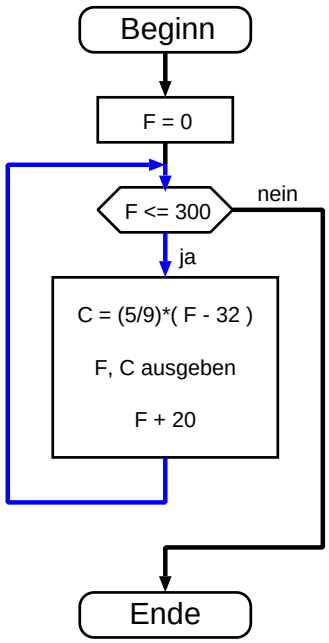


Abbildung 1.2.: Programmablaufplan „Umrechnungstabelle“

Verwendung einer `while`-Schleife

Liste 1.8: Umrechnungstabelle $°F$ nach $°C$ (Verwendung einer `while`-Schleife)

```
1  #include  <stdio.h>
2
3  int main  ()
4  {
5    double    fahr, celsius;                    /* Vereinbarungen */
6    int       unten, oben, schritt;
7
8    unten   =   0;                              /* Tabellenanfang */
9    oben    = 300;                              /* Tabellenende   */
10   schritt =  20;                              /* Schrittweite   */
11   fahr    = unten;
12
13   /* ---------- Programmtitel, Tabellenkopf -------------------------- */
14   printf("\nUmwandlung von Grad Fahrenheit in Grad Celsius");
15   printf("\n\nGrad Fahrenheit\tGrad Celsius\n");
16
17   /* ---------- Tabellenzeilen ausgeben ------------------------------ */
18   while ( fahr <= oben ) {
19     celsius = (5.0/9.0)*(fahr - 32.0);
20     printf( "%3.0lf\t\t%6.1lf\n", fahr, celsius );
21     fahr    = fahr + schritt;
22     }
23   return 0;
24 }
```

In Liste 1.8 ist das zugehörige Programm angegeben. Es enthält einige neue Elemente, die nachfolgend erläutert werden.

Zeile 6 Obwohl zur Lösung der eigentlichen Aufgabe nur zwei Temperaturvariablen erforderlich wären, verbessert die Einrichtung der Hilfsvariablen `unten`, `oben`, `schritt` die Lesbarkeit und die Änderbarkeit des Programmes.

Zeile 8-11 Anfangswertzuweisungen an die Hilfsvariablen.

Zeile 18 `while`-Schleife zur wiederholten Berechnung einer Tabellenzeile. Die Variable `fahr` steuert die Schleife. Sie wird gegen die Obergrenze in der Variablen `oben` verglichen. Der nachfolgende Schleifenkörper wird betreten, wenn die Bedingung `fahr <= oben` wahr ist. Bei der ersten Auswertung besitzt `fahr` den Wert 0, `oben` den Wert 300; die Bedingung ist also wahr. Wenn die Bedingung bei einem der nächsten Durchläufe falsch wird, wird die Programmausführung hinter dem `while`-Block fortgesetzt. Die allgemeine Form einer `while`-Anweisung lautet:

```
while( Bedingung )
{
  Schleifenkörper
}
```

Zeile 19 Die Umrechnungsformel.

Zeile 20 Ausgabe einer Tabellenzeile.

Zeile 21 Der Wert der Variablen `fahr` wird um die Tabellenschrittweite in `schritt` erhöht. Durch fortgesetzte Erhöhung wird nach einigen Durchläufen der Vergleich in Zeile 18 den Wert „falsch“ ergeben und die Schleife damit beenden.

Diese Anweisung darf trotz des Gleichheitszeichens nicht als mathematische Gleichung aufgefaßt werden ! Es handelt sich hier um eine *Zuweisung*: die rechte Seite der Zuweisung wird berechnet und der ermittelte Wert wird anschließend der Variablen auf der linken Seite zugewiesen.

Man erkennt, daß die Bedingung, die die Schleife steuert, nur falsch wird, wenn sie durch geeignete Maßnahmen innerhalb der Schleife beeinflußt wird.

Verwendung einer `for`-Schleife

Die Organisation der Wiederholung mit einer `while`-Schleife erfordert offensichtlich drei Maßnahmen:

- Zuweisung eines Startwertes an die Steuervariable der Schleife.
- Überprüfung einer sogenannten Abbruchbedingung, die über die weitere Durchführung der Schleife entscheidet.
- Veränderung der Steuervariablen um nach der gewünschten Anzahl von Durchläufen die Schleife zu beenden.

Die `for`-Schleife faßt diese drei Maßnahmen in einer Steueranweisung zusammen. Liste 1.9 zeigt die gleiche Lösung wie oben unter Verwendung einer `for`-Schleife.

Liste 1.9: Umrechnungstabelle $°F$ nach $°C$ (Verwendung einer `for`-Schleife)

```
1  #include  <stdio.h>
2
3  int main  ( )
4  {
5    int       fahr;                                /* Vereinbarungen */
6    double    celsius;
7    int       unten, oben, schritt;
8
9    unten   =   0;                                 /* Tabellenanfang */
10   oben    = 300;                                 /* Tabellenende   */
11   schritt =  20;                                 /* Schrittweite   */
12
13   /* ---------- Programmtitel, Tabellenkopf ------------------------- */
14   printf("\nUmwandlung von Grad Fahrenheit in Grad Celsius");
15   printf("\n\nGrad Fahrenheit\tGrad Celsius\n");
16
17   /* ---------- Tabellenzeilen ausgeben ----------------------------- */
18   for ( fahr = unten; fahr <= oben; fahr = fahr+schritt ) {
19     celsius = (5.0/9.0)*(fahr - 32.0);
20     printf( "%3d\t\t%6.1lf\n", fahr, celsius );
21     }
22
23   return 0;
24 }
```

Die allgemeine Form einer `for`-Anweisung lautet:

```
for( Ausdruck-1 ;  Ausdruck-2 ;  Ausdruck-3 )
{
  Schleifenkörper
}
```

Ausdruck-1 Anfangswertzuweisung für die Steuervariable der Schleife; diese Anweisung wird genau *einmal vor Beginn der Schleife* ausgeführt.

Ausdruck-2 Abbruchbedingung; die Schleife läuft so lange, wie diese Bedingung wahr ist. Die Bedingung wird *vor jedem Schleifendurchlauf* geprüft.

Ausdruck-3 Veränderung des Schleifenzählers; die Veränderung wird *nach jeden Schleifendurchlauf* durchgeführt.

`for`-Schleifen kommen häufig vor und sind besonders bei der Bearbeitung von Feldern sehr nützlich. Der Vorteil besteht in der Übersichtlichkeit. Alle Steueranweisung sind im Schleifenkopf zusammengefaßt.

Verwendung einer `if`-Anweisung

Zur Gestaltung von Programmabläufen sind neben Wiederholung auch Verzweigungen erforderlich. Abhängig von der Auswertung einer oder mehrerer Bedingungen sollen Anweisungen ausgeführt werden oder eben nicht. Eine Möglichkeit dazu ist die Verwendung einer `if`-Anweisung. Die allgemeine Form lautet:

```
if( Bedingungen )
{
  Anweisungsteil-1
}
else
{
  Anweisungsteil-2
}
```

- Wenn die Bedingung wahr ist wird der Anweisungsteil-1 ausgeführt, andernfalls der Anweisungsteil-2.
- Wenn ein Anweisungsteil aus nur einer Anweisung besteht, können die geschweiften Klammern um diese Anweisung fehlen.
- Der `else`-Teil kann vollständig fehlen (einfache bedingte Anweisung).

In den nachfolgenden Zeilen wird der Variablen `y` der Betrag von `x` zugewiesen, $y = |x|$. Weil in jedem Anweisungsteil nur eine Anweisung steht, können die geschweiften Klammern fehlen.

```
if( x < 0.0 )
  y = -x;
else
  y =  x;
```

Um den Wert von von `x` durch seinen eigenen Betrag zu ersetzen, $x = |x|$, reicht eine einfache bedingte Anweisung:

```
if( x < 0.0 )
  x = -x;
```

Zur Darstellung mehrstufiger Entscheidungen können `if-else`-Anweisungen geschachtelt werden.

1.8. Einfache Funktionen

Mit Hilfe von Schleifen und Verzweigungen wird die Ablauflogik eines Programmes gestaltet. Zur Gliederung größerer Programmumfänge ist es notwendig, Programmstücken, die eine abgeschlossene Teilaufgabe lösen und durch wenige Parameter steuerbar sind, zu geschlossenen Einheiten zusammenzufassen. Diese Einheiten heißen Funktionen oder Prozeduren und sind das wichtigste Gliederungsmittel höherer Programmiersprachen auf mittlerer Ebene. Darüber hinaus besteht bei größeren Projekten noch die Möglichkeit, den Code auf mehrere Dateien zu verteilen.

Die Verwendung von Funktionen hat weitere wesentliche Vorteile. Der Code kann mehrfach verwendet werden, da eine einmal geschriebene Funktion viele Male von unterschiedlichen Programmstellen aus aufgerufen werden kann. Die Lösung der Teilaufgabe muß außerdem nur einmal programmiert werden. Bei Änderungen werden diese nach einer Neuübersetzung aber an allen Aufrufstellen wirksam.

Die Schreib- und Bezeichnungsweise für Funktionen in höheren Programmiersprachen lehnt sich an den Sprachgebrauch der Mathematik an. Die Funktion einer Unbekannten,

$$f(x) = x^2 - 27 \cdot x + 13$$

stellt eine Berechnungsvorschrift dar. Im Anwendungsfall muß für die unabhängige Größe x ein Zahlenwert eingesetzt und die rechte Seite der Gleichung ausgewertet werden.

$$f(3) = 3^2 - 27 \cdot 3 + 13$$

In gleicher Weise kann die bereits bekannte Temperaturumrechnung als Funktion $c(fahrenheit)$ geschrieben und auch programmiert werden:

$$c(fahrenheit) = \frac{5}{9}(fahrenheit - 32)$$

Die Größe $fahrenheit$ ist die unabhängige Variablen oder das *Argument der Funktion*. Durch Einsetzen eines Fahrenheitwertes erhalten wir den Wert der abhängigen Größe $celsius$.

In gleicher Weise verfahren höhere Programmiersprachen. Eine Berechnungsvorschrift oder ein beliebiger anderer Algorithmus kann als Funktion geschrieben werden. Man spricht von der *Definition einer Funktion*. Die Parameter, die zur Berechnung erforderlich sind, müssen vorläufige Namen erhalten, um dem Algorithmus überhaupt formulieren zu können. Das entspricht den unabhängigen Variablen in den gerade genannten mathematischen Beispielen.

Allein durch die Definition einer Funktion findet noch keine Berechnung statt. Dazu muß die Funktion aufgerufen werden. Beim Aufruf müssen, genau wie bei der Auswertung einer mathematischen Funktion, konkrete Werte für die unabhängigen Größen, die Funktionsparameter, übergeben werden. Ein derartiger Aufruf liefert dann das abhängige Berechnungsergebnis. Bei einer einfachen ***C***-Funktion ist das der sogenannte *Rückgabewert*.

Das in Liste 1.10 gezeigte Programm erstellt dieselbe Umrechnungstabelle wie die bisherigen Fassungen. Zur Auswertung der Umrechnungsformel wird nun aber eine Funktion eingesetzt.

Liste 1.10: Umrechnungstabelle $^\circ F$ nach $^\circ C$ (Verwendung einer Funktion)

```
1  #include  <stdio.h>
2
3  /*------------------------------------------------------------------------
4   *  Definition der Funktion celsius
5   *-----------------------------------------------------------------------*/
6  double  celsius ( double fahrenheit )
7  {
8    double  ergebnis;                              /* Hilfsvariable        */
9    ergebnis  = (5.0/9.0)*(fahrenheit-32.0);       /* Umrechnungsformel    */
10   return  ergebnis;                              /* Ergebnis zurückgeben */
11 }
12
13 /*------------------------------------------------------------------------
14  *  Hauptprogramm
15  *-----------------------------------------------------------------------*/
16 int main  ( )
17 {
18   double    fahr, cels;                          /* Vereinbarungen */
19   int       unten, oben, schritt;
20
21   unten   =   0;                                 /* Tabellenanfang */
22   oben    = 300;                                 /* Tabellenende   */
23   schritt =  20;                                 /* Schrittweite   */
24
25   /* ---------- Programmtitel, Tabellenkopf ------------------------- */
26   printf("\n\t*****  Umwandlung von Grad Fahrenheit in Grad Celsius  *****");
27   printf("\n\n\tGrad Fahrenheit\tGrad Celsius\n");
28
29   /* ---------- Tabellenzeilen ausgeben ----------------------------- */
30   for ( fahr=unten; fahr<=oben; fahr=fahr+schritt ) {
31     cels  = celsius( fahr );                     /* Aufruf der Funktion celsius */
32     printf( "\t\t%7.0lf\t%12.1lf\n", fahr, cels );
33     }
34
35   return 0;
36 }
```

Zeile 6-11 Die Definition der Funktion `celsius`.

Zeile 6 Der Funktionskopf.

Der Funktionsname lautet `celsius`.

Die Funktion besitzt den Rückgabetyp `double`, das heißt sie gibt als Berechnungsergebnis eine `double`-Größe zurück. Deshalb steht das Schlüsselwort `double` vor dem Funktionsnamen.

Der formale Parameter `fahrenheit` wird beim Aufruf durch einen Zahlenwert ersetzt. Dieser Zahlenwert wird dann bei der Durchführung der Berechnung in Zeile 9 tatsächlich verwendet. Der Parameter `fahrenheit` besitzt den Datentyp `double`.

Zeile 8 Vereinbarung einer Hilfsvariablen `ergebnis`. Diese Variable existiert nur innerhalb dieser Funktion.

Zeile 9 Die Umrechnung.

Zeile 10 Die `return`-Anweisungen veranlaßt den Rücksprung zur Aufrufstelle. Hier wird die Funktion verlassen. Diese Anweisung veranlaßt außerdem die Rückgabe des errechneten Temperaturwertes, der in `ergebnis` gespeichert ist.

Zeile 31 Hier wird die Funktion `celsius` auf der rechten Seite einer Zuweisung aufgerufen. An dieser Stelle wird das Hauptprogramm bei jedem Schleifendurchlauf verlassen. Die Programmausführung springt in die Funktion `celsius` und veranlaßt die Berechnung für den gerade in der Schleife aktuellen Wert von `fahr`. Nach Beendigung der Funktion wird der errechnete Wert des Aufrufs nach links an die Größe `cels` zugewiesen.

Die Definition einer ***C***-Funktion besitzt folgenden allgemeinen Aufbau:

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

Es gelten folgende Regeln:

- Der Funktionsname ist frei wählbar (Groß-, Kleinbuchstaben, Ziffern und Unterstriche sind zulässig. Das erste Zeichen darf keine Ziffer sein).
- Ein Rückgabetyp muß angegeben werden.
- Die Parameterliste enthält formale Parameter, das heißt Platzhalter für die beim Aufruf tatsächlich zu übergebenden Werte. Die einzelnen Parameter werden durch Kommata getrennt.
- Für jeden Parameter ist der Datentyp anzugeben.
- Die Parameterliste kann leer sein; die runden Klammern müssen aber in jedem Fall da sein.
- Vereinbarungen und Anweisungen können fehlen.

1.9. Felder und Zeichenketten

Felder

Felder dienen zum Abspeichern von mehreren Variablen mit *demselben Datentyp und Namen.* Die Unterscheidung der einzelnen Feldelemente geschieht durch Indices.

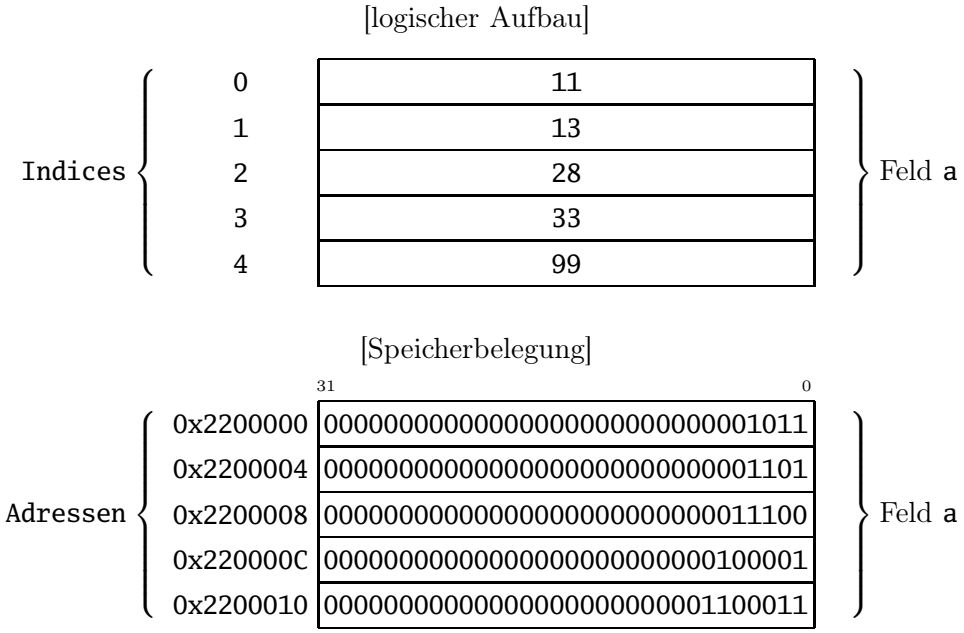


Abbildung 1.3.: `int`-Feld mit 5 Elementen

Zur Vereinbarungen eines `int`-Feldes mit Namen `a` und 5 Elementen wird die folgende Anweisung verwendet:

```
int  a[5];
```

Die Vereinbarung entspricht zunächst der einer einfachen Variablen vom Typ `int`. Zusätzlich wird aber die Anzahl der Feldelemente in eckigen Klammern nachgestellt. Mit dieser Vereinbarung entsteht ein Feld, welches aus 5 einzelnen `int`-Variablen mit folgenden Namen besteht:

```
a[0]   a[1]   a[2]   a[3]   a[4]
```

Der Index des ersten Feldelementes ist immer 0, der Index des letzten Feldelementes entspricht der Feldlänge minus 1. Aus der Sicht des Programmierers ist ein Feld also eine Aneinandereihung von Speicherplätzen gleichen Namens, deren Numerierung mit 0 beginnt.

Tatsächlich liegen die Dateninhalte auch in aufeinanderfolgenden Speicherplätzen im Hauptspeicher. Die Abbildung 1.3 zeigt die Verhältnisse für das Feld `a`. Ein einzelnes Feldelement ist in jeder Beziehung wie eine einfache Variable desselben Datentyps zu behandeln (Ein-/Ausgabe, Zuweisung, Verwendung in Rechenausdrücken und so weiter). In Liste 1.11 wird die Verwendung eines Feldes in einem vollständigen Programm gezeigt.
Das Feld `a` wird in der ersten Schleife mit Quadratzahlen gefüllt. In der zweiten Schleife werden diese in Reihen zu 10 ausgegeben. An dem Beispiel sieht man auch, daß als Feldelementindex nicht nur Konstanten, sondern auch Variablen und sogar Rechenausdrücke verwendet werden dürfen.

Liste 1.11: Verwendung eines Feldes

```
1  #include    <stdio.h>
2
3  #define     N   100                              /* Feldumfang    */
4
5  int main (void)
6  {
7    int   a[N];                                    /* Feld          */
8    int   i, n = N;
9
10   printf("\nFeld mit Quadratzahlen belegen und ausgeben\n");
11
12   for ( i=0; i<n; i=i+1 )                        /* Feld füllen   */
13     a[i]  = i*i;
14
15   for ( i=0; i<n; i=i+1 )                        /* Feld ausgeben */
16   {
17     if(i%10==0) printf("\n");     /* Zeilenvorschub nach jeweils 10 Zahlen */
18     printf(" %6d", a[i] );
19   }
20   printf("\n\n");
21
22   return 0;
23 }       /* ----------  end of function main  ---------- */
```

Zeichenketten

Für die Handhabung von Einzelzeichen und Zeichenketten steht in *C* der Basisdatentyp `char` zur Verfügung. Eine Variable von Typ `char` kann ein einzelnes Zeichen aufnehmen. Die Vereinbarung von Variablen geschieht wie folgt:

```
char  z1, z2, zeichen;
```

Einer `char`-Variablen kann durch direkte Zuweisung oder durch Einlesen ein Zeichen zugewiesen werden. Zeichenkonstanten sind dabei grundsätzlich in einfache, senkrechte Anführungszeichen `' '` einzurahmen, damit sie der Compiler von Variablennamen unterscheiden kann. Die folgende Zuweisung speichert den Kleinbuchstaben `b` in der Variablen `zeichen`.

```
zeichen = 'b';
```

Die Zuweisung von Steuerzeichen ist ebenfalls möglich. Hier muß wieder auf die Ersatzdarstellung zurückgegriffen werden.

```
zeichen = '\n'
```

Die Variable `zeichen` enthält nun die Codierung des Zeilenvorschubes. Bei der Ausgabe der Variablen wird der Zeilenvorschub auch wirksam. Die Ausgabe eines einzelnen Zeichens geschieht mit dem Teilformat `%c`. Die Angabe von Flags und einer Stellenanzahl ist möglich (siehe Tabelle 1.5).

Die erste der folgenden `printf`-Anweisungen gibt die Zeichen `'␣x'` aus. Das Zeichen `x` erscheint in 2 Stellen rechtsbündig. Die zweite `printf`-Anweisungen gibt die Variable als Hexadezimalzahl in 4 Stellen rechtsbündig aus: `'␣␣78'`.

```
zeichen = 'x';
printf("%2c\n", zeichen );     /* Zeichenausgabe       */
printf("%4x\n", zeichen );     /* hexadezimale Ausgabe */
```

Die letzte Ausgabe weist auf eine unerwartete Eigenschaft des Datentyps `char` hin: die `char`-Größen sind ihrer Natur nach ganzzahlige Typen, mit denen auch gerechnet[3] werden kann!

Das Anlegen von Feldern ist wie bei den anderen Datentypen möglich:

```
char  puffer[100];
```

Die Ausgabefunktionen (zum Beispiel `printf`) und die Standardfunktionen zur Handhabung von Zeichenketten erwarten, daß eine Zeichenkette mit dem ASCII-Steuerzeichen `NUL` abgeschlossen ist. Das Steuerzeichen `NUL` ist eine binäre Null, die als `char`-Konstante durch `'\0'` dargestellt wird. Sie darf nicht mit dem Zeichen `0` für die Ziffer 0, ASCII-Codierung 48 (dezimal), verwechselt werden[4].

Funktion	Beschreibung
`int isalnum ( int c );`	prüft, ob ein Zeichen alphanumerisch ist
`int isalpha ( int c );`	prüft, ob ein Zeichen ein Buchstabe ist
`int iscntrl ( int c );`	prüft, ob ein Zeichen ein Kontrollzeichen ist
`int isdigit ( int c );`	prüft, ob ein Zeichen eine Ziffer ist
`int islower ( int c );`	prüft, ob ein Zeichen ein Kleinbuchstabe ist
`int isprint ( int c );`	prüft, ob ein Zeichen ein abdruckbares Zeichen ist
`int ispunct ( int c );`	prüft, ob ein Zeichen ein abdruckbares Zeichen, aber kein Leerzeichen und kein alphanumerisches Zeichen ist
`int isspace ( int c );`	prüft, ob ein Zeichen ein Leerzeichen oder ein Vorschubsteuerzeichen ist
`int isupper ( int c );`	prüft, ob ein Zeichen ein Großbuchstabe ist
`int isxdigit ( int c );`	prüft, ob ein Zeichen eine Hexadizimalziffer ist
`int toupper ( int c );`	wandelt Kleinbuchstaben in Großbuchstaben um
`int tolower ( int c );`	wandelt Großbuchstaben in Kleinbuchstaben um

Tabelle 1.6.: Klassifizierungsfunktionen für Einzelzeichen (Auswahl; `include`-Datei `ctype.h`)

Funktion	Beschreibung
`int getchar ( void );`	liest ein einzelnes Zeichen von der Standardeingabe (Tastatur)
`int putchar ( int c );`	gibt ein einzelnes Zeichen auf die Standardausgabe aus (Bildschirm)

Tabelle 1.7.: Ein-/Ausgabefunktionen für Zeichenketten (Auswahl; `include`-Datei `stdio.h`)

Wenn im eigenen Programm Zeichenketten in einem Feld aufgebaut werden, ist auf diesen richtigen Abschluß zu achten. Das heißt auch, daß ein Feld mindestens einen Platz mehr umfassen muß als die längste zu verarbeitende Zeichenkette lang ist. Das Abschlußzeichen wird

[3] Wegen der beschränkten internen Darstellung ist der Umfang des Zahlenbereiches allerdings gering (zum Beispiel 8 Bit = 1 Byte, damit sind $2^8 = 256$ Werte möglich).

[4] Die ASCII-Codierung ist in vielen Programmierbüchern wiedergegeben. Auf ***Unix***-/***Linux***-Systemen kann sie in der Systemdokumentation zum Beispiel durch `man ascii` nachgeschlagen werden. Heute hat sich die Verwendung von UNICODE durchgesetzt. Die ASCII-Zeichen bilden eine Untermenge davon.

auch zur Erkennung des Zeichenkettenendes verwendet. Mit Hilfe einer Schleife kann zum Beispiel solange von vorne durch die Feldelemente gegangen werden, bis ein Nullzeichen gefunden wird (siehe Liste 1.12).
Zum Umgang mit Zeichen und Zeichenketten gibt es eine Reihe von Funktionen in der *C*-Standardbibliothek. Die beiden Ein-/Ausgabefunktionen in Tabelle 1.7 erlauben die Ein- und Ausgabe einzelner Zeichen. Die Verwendung wird in dem Beispielprogamm in Liste 1.12 gezeigt.
Tabelle 1.6 enthält eine Auswahl an Klassifizierungsfunktionen, die zum Beispiel in [Han06] beschrieben sind.

Liste 1.12: Klein- und Großbuchstaben abzählen

```
1  #include    <stdio.h>
2  #include    <ctype.h>
3
4  #define     MAXSTRING     256               /* Größe des Eingabepuffer         */
5
6  int main ( )
7  {
8    char  c, text[MAXSTRING];                 /* Hilfsvariable, Eingabepuffer    */
9    int   i, sumkl, sumgr;                    /* Schleifenzähler, Buchst.zähler  */
10
11   printf("\n\t*****  Klein- und Großbuchstaben abzählen  *****");
12
13   printf("\n\n\tGeben Sie eine Textzeile ein: ");
14
15   i = 0;
16   while( (c=getchar()) != '\n' )            /* Klammerung erforderlich! */
17   {
18     text[i] = c;
19     i = i+1;
20   }
21   text[i] = '\0';                           /* Textendezeichen setzen   */
22
23   printf("\n\n\t       Der eingegebene Text : ");
24   for(i=0; text[i] != '\0'; i=i+1)          /* Text ausgeben            */
25     putchar(text[i]);
26
27   sumgr = sumkl = 0;
28   for( i=0; text[i] != '\0'; i=i+1 )        /* Text untersuchen         */
29   {
30     if( islower(text[i]) )  sumkl = sumkl + 1;
31     if( isupper(text[i]) )  sumgr = sumgr + 1;
32   }
33
34   printf("\n\n\t%d Kleinbuchstaben / %d Großbuchstaben\n\n", sumkl, sumgr );
35   return 0;
36 }       /* ----------  end of function main  ---------- */
```

Liste 1.12 zeigt ein Programm, in welchem ein beliebiger einzeiliger Text in einen Puffer (`char`-Feld) eingelesen wird. Anschließend wird jedes Zeichen im Puffer daraufhin überprüft, ob ein Klein- oder ein Großbuchstabe vorliegt.

Zeile 16 Mit Hilfe der Funktion `getchar` werden solange einzelne Zeichen von der Tastatur eingelesen, bis das Zeichen `'\n'` gelesen wird. Dieses Zeichen wird von der Eingabetaste

(return-Taste) erzeugt, mit der die Texteingabe an das Programm übergeben wird. In der Abbruchbedingung der `while`-Schleife

```
while( (c=getchar()) != '\n' )
```

geschehen mehrere Dinge. Zunächst wird das von `getchar` gelesene Zeichen an die Variable `c` zugewiesen. Diese Zuweisung ist geklammert. Der Wert dieser Klammer entspricht dem zuletzt zugewiesenen Zeichenwert. Deshalb kann der Wert dieser Klammer mit der Zeichenkonstante `'\n'` auf Ungleichheit (Vergleichsoperator `!=`) verglichen werden. Die Schleife läuft solange, wie das eingelesene Zeichen kein Zeilenvorschub ist.

Zeile 18 Der gelesene Zeichenwert wird im Feld `text` abgelegt.

Zeile 19 Der Feldindex `i` wird um 1 erhöht. Er zeigt immer auf den nächsten freien Feldplatz.

Zeile 21 Das Textendezeichen `'\0'` wird als Textabschluß eingesetzt.

Zeile 27 Die Zähler für die Groß- und Kleinbuchstaben werden auf Null gesetzt (Mehrfachzuweisung).

Zeile 28 Die `for`-Schleife läuft von Index 0 ab durch das Textfeld (Variable `i`). Die Schleife läuft solange, bis das Abschlußzeichen erreicht ist (`text[i] != '\0'`).

Zeile 30,31 Mit Hilfe der Klassifizierungsfunktionen `islower` und `isupper` werden Groß- und Kleinbuchstaben erkannt und gegebenenfalls der jeweilige Zähler erhöht.

Zeile 34 Ergebnisausgabe.

Wenn der eingegebene Text zum Beispiel

```
Dies␣ist␣ein␣Text.
```

lautete (␣ : Leerzeichen), dann liegt nach der `while`-Schleife folgende Belegung des Feldes `text` vor:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	...	254	255
D	i	e	s	␣	i	s	t	␣	e	i	n	␣	T	e	x	t	.	NUL	?	...	?	?

Nach dem Punkt des Textes steht in der Indexposition 18 das Abschlußzeichen `NUL` (binäre Null). Die Belegung der nachfolgenden Feldplätze ist unbestimmt und deshalb mit einem Fragezeichen markiert.

1.10. Dateieingabe und Dateiausgabe

Dialogeingaben und -ausgaben sind dann geeignet, wenn einige wenige Werte einzugeben oder Ergebnisse auszugeben sind. Für größere Datenumfänge muß aus Aufwands- und Sicherheitsgründen (Vertippen!) die Ein- und Ausgabe aus beziehungsweise in eine Datei erfolgen.

Umlenkung des Ein- und Ausgabestromes

Wenn auf der Konsole gearbeitet wird (***Unix***, ***Linux***, ***DOS***, ...) besteht die Möglichkeit, die Programmeingaben und -ausgaben umzulenken.

Der Aufruf

```
prog1 > ausgabe.dat
```

lenkt alle Ausgaben des Programmes `prog1` in die Datei `ausgabe.dat` um. Wenn die Ausgabedatei nicht vorhanden ist, wird sie angelegt, wenn sie vorhanden ist, wird sie überschrieben. Entsprechend können alle Eingaben aus einer Datei gelesen werden:

```
prog2 < eingabe.dat
```

Die Verbindung der beiden Vorgänge ist ebenfalls möglich:

```
prog3 < eingabe.dat > ausgabe.dat
```

Der Nachteil dieser Vorgehensweise ist, daß eine Dialogführung nicht mehr möglich ist, weil nun *alle* Eingaben aus einer Datei kommen beziehungsweise *alle* Ausgaben in eine Datei geschrieben werden. Der Vorteil ist, daß insbesondere umfangreiche Ausgabedaten einfach aufgefangen werden können, um sie zum Beispiel anschließend in ein Plot-Programm oder eine Tabellenkalkulation zu importieren.

Lesen und Schreiben von Dateien mit `fscanf` und `fprintf`

Eine bequemere Möglichkeit, die zudem auch für mehrere Dateien und für Nichtkonsolprogramme anwendbar ist, ist das Lesen und Schreiben von Dateien direkt aus dem Programm. Dazu stehen vier grundlegende Funktionen zur Verfügung, die in Tabelle 1.8 angegeben sind.

Funktion	Beschreibung
`FILE *fopen ( const char *path, const char *mode );`	Datei öffnen
`int fclose ( FILE *stream);`	Datei schließen
`int fscanf ( FILE *stream, const char *format, ... );`	Lesen
`int fprintf ( FILE *stream, const char *format, ... );`	Schreiben

Tabelle 1.8.: Formatierte Dateieingabe und Dateiausgabe (`stdio.h`)

Öffnen und Schließen einer Datei

In einem Programm können mehrere Dateien gleichzeitig verwendet werden. Bevor das geschehen kann, muß jede Datei geöffnet werden, das heißt der Zugriff muß beim Betriebssystem des Rechners angemeldet werden. Auch die Lese- und Schreiboperationen können nur über Systemaufrufe erfolgen. Diese verbergen sich allerdings hinter den entsprechenden ***C***-Funktionen der Standardbibliothek (`fprintf`, und so weiter).

Der erste Schritt zur Verwendung einer Datei ist also das Öffnen. Dazu sind mindestens folgende Zeilen erforderlich (hier in einem Hauptprogramm):

```
#include <stdio.h>                          /* Standardeingabe/-ausgabe    */

int main (void)
{
  FILE    *eingabe;                         /* Zeiger auf die Eingabedatei */

  eingabe = fopen ( "vektor.dat", "r" );    /* Datei öffnen                */

  /* ...  weitere Anweisungen  ... */

  fclose( eingabe );                        /* Eingabedatei schließen      */

  return 0;
}
```

Der Zugriff auf Dateien erfolgt über Zeiger. Zeiger sind Variablen die Speicheradressen enthalten können. Sie werden in einem späteren Kapitel vorgestellt. Durch die Zeile

```
FILE    *eingabe;
```

wird ein derartiger Dateizeiger eingerichtet. Der Datentyp `FILE` ist in `stdio.h` bereits definiert und kann ohne weiteres verwendet werden. Der Name `eingabe` ist frei gewählt. Die gewünschte Datei wird durch den Aufruf der Funktion `fopen`

```
eingabe = fopen ( "vektor.dat", "r" );
```

geöffnet. `fopen` erhält als ersten Parameter den Namen der zu öffnenden Datei als Zeichenkette. Der zweite Parameter, ebenfalls eine Zeichenkette, enthält den sogenannten Öffnungsmodus. Die Angabe `"r"` steht für Lesen (Tabelle 1.9 enthält die möglichen Modi).

Modus	Bedeutung
`r`	Lesen (**r**ead)
`r+`	Lesen und Schreiben
`w`	Schreiben (**w**rite)
`w+`	Lesen und Schreiben
`a`	Anhängen (**a**ppend)
`a+`	Lesen und Anhängen

Beim Öffnen für Schreiben und Lesen (die ersten vier Einträge) beginnt die Ein- oder Ausgabe am Dateianfang, beim Anhängen am Dateiende.

Tabelle 1.9.: Modi zum Öffnen von Dateien

Im Erfolgsfall liefert `fopen` eine Speicheradresse zurück, die hier an die Variable `eingabe` zugewiesen wird. Die beiden Funktionen `fscanf` und `fprintf` verwenden diese Zeiger (siehe unten). Ein geöffnete Datei wird mit der Anweisung

```
fclose( eingabe );
```

wieder geschlossen. Das kann geschehen, sobald sie nicht mehr benötigt wird. Als Parameter für `fclose` ist der entsprechende Dateizeiger zu übergeben.

Schreiben in eine Datei

Zum Schreiben in eine Datei kann die Funktion `fprintf` verwendet werden. Sie entspricht der Funktion `printf`. Vor der Formatzeichenkette steht aber als zusätzlicher Parameter der Zeiger auf eine geöffnete Datei. Die folgende Zeile gibt den Wert der Variablen `laenge` formatiert in die Datei aus, auf die der Zeiger `ausgabe` zeigt:

```
fprintf( ausgabe, "\n%f", laenge );
```

Die Formatierungsmöglichkeiten entsprechen genau denen von `printf`, so daß hier nichts Neues hinzu kommt.

`fprintf(` ***Dateizeiger***, ***Format***, ***Argumentliste*** `);`

Lesen aus einer Datei

Erwartungsgemäß gilt das für `fprintf` Gesagt auch für das Lesen aus einer offenen Datei mit Hilfe der Funktion `fscanf`. Auch hier muß als erster Parameter der Zeiger auf die Eingabedatei angegeben werden. Darauf folgt die Formatbeschreibung und die Liste der einzulesenden Größen.

```
fscanf( eingabe, "%lf", &umfang );
```

Für die Formatbeschreibung und die Verwendung des Adreßoperators gilt das Gleiche wie bei der Funktion `scanf`.

`fscanf(` ***Dateizeiger***, ***Format***, ***Argumentliste*** `);`

Fehlerbehandlung

Vorkehrungen gegen mögliche Fehlerfälle sind bei professioneller Arbeitsweise selbstverständlich. So muß zum Beispiel nach dem Öffnen einer Datei geprüft werden, ob die Datei tatsächlich offen ist. Wenn die Datei nicht geöffnet werden konnte (falscher Dateiname, Festplatte voll, bereits zu viele Dateien offen, ...) besitzt der Dateizeiger den besonderen Wert `NULL`. Der genaue Wert von `NULL` interessiert im allgemeinen nicht, da er zur Überprüfung nur in einen Vergleich eingeht. Die Größe `NULL` ist in `stdio.h` als `define`-Makro definiert. Eine Überprüfung kann zum Beispiel wie folgt aussehen:

```
eingabe = fopen ( "vektor.dat", "r" );
if ( eingabe == NULL ) {
  printf("\nDatei vektor.dat konnte nicht geöffnet werden !\n");
  exit(1);                          /* Programm hier abbrechen */
}
```

In der `if`-Bedingung wird überprüft, ob der Zeiger `eingabe` den Wert `NULL` hat. Wenn das der Fall ist, wird die nachfolgende Fehlermeldung ausgegeben und das Programm mit dem Aufruf `exit(1)` abgebrochen.

Ein vollständiges Beispiel

Liste 1.13: Formatierte Dateieingabe und Dateiausgabe

```
1  #include <stdio.h>
2  #include <stdlib.h>                          /* wegen exit()                */
3
4  #define   N   100                            /* Feldumfang                  */
5
6  int main (void)
7  {
8    FILE    *eingabe;                          /* Zeiger auf die Eingabedatei */
9    FILE    *ausgabe;                          /* Zeiger auf die Ausgabedatei */
10   double  vektor [N];                        /* Vektor; Eingabedaten        */
11   int     i, n;                              /* Hilfsgrößen                 */
12
13   /* ---------- Eingabedatei öffnen ----------  */
14   eingabe = fopen ( "vektor.dat", "r" );
15   if ( eingabe == NULL ) {
16     printf("\nDatei vektor.dat konnte nicht geöffnet werden !\n");
17     exit(1);                                 /* Programm hier abbrechen     */
18   }
19
20   /* ---------- Eingabedatei lesen ----------  */
21   fscanf( eingabe, "%d", &n );               /* Anzahl Vektorelemente lesen */
22   for ( i=0; i<n; i=i+1 )
23     fscanf( eingabe, "%lf", &vektor[i] );
24
25   fclose( eingabe );                         /* Eingabedatei schließen      */
26
27   /* ---------- Ausgabedatei öffnen ----------  */
28   ausgabe = fopen ( "vektor.aus", "w" );
29   if ( ausgabe == NULL ) {
30     printf("\nDatei vektor.aus konnte nicht geöffnet werden !\n");
31     exit(1);                                 /* Programm hier abbrechen     */
32   }
33
34   /* ---------- vektor[] bearbeiten: Werte aller Elemente verdoppeln */
35   for ( i=0; i<n; i=i+1 )
36     vektor[i] = 2.0*vektor[i];
37
38   /* ---------- Ausgabedatei schreiben  ----------  */
39   fprintf( ausgabe, "%d", n );               /* Anzahl der Vektorelemente   */
40   for ( i=0; i<n; i=i+1 )
41     fprintf( ausgabe, "\n%5.1lf", vektor[i] );
42
43   fclose( ausgabe );                         /* Ausgabedatei schließen      */
44
45   return 0;
46 }       /* ----------  end of function main  ---------- */
```

Die Liste 1.13 zeigt die bisher dargestellten Möglichkeiten in einem vollständigen Programm. Aus einer Eingabedatei `vektor.dat` (Liste 1.14) wird zunächst die Anzahl `n` der weiterhin einzulesenden Vektorelemente eingelesen. Mit diesem Wert werden in der nachfolgenden Schleife die Elemente eingelesen. Der Wert aller Vektorelemente wird daraufhin verdoppelt. Anschließend wird der Vektor in die Ausgabedatei `vektor.aus` geschrieben (Liste 1.15).

Liste 1.14: Eingabedatei `vektor.dat`

```
1  5
2  1.1
3  2.3
4  3.5
5  5.4
6  6.2
```

Liste 1.15: Ausgabedatei `vektor.aus`

```
1  5
2    2.2
3    4.6
4    7.0
5   10.8
6   12.4
```

Lesen einer Datei unbekannter Länge

Wenn die Länge der einzulesenden Datei nicht wie in Liste 1.13 als erster Wert in der Datei selbst steht, sondern der Wunsch besteht, Dateien beliebiger Länge fortlaufend bis zu ihrem Ende einzulesen, dann hilft folgende Vorgehensweise. Die Funktion `fscanf` führt nicht nur die Einlesevorgänge für die in der Argumentliste aufgeführten Variablen durch, sie gibt auch einen Wert als Ergebnis des Funktionsaufrufes zurück, der über Erfolg oder Mißerfolg Auskunft gibt. Dieser Rückgabewert könnte in Form einer Zuweisung entgegengenommen werden,

```
ergebnis = fscanf( eingabe, "%lf", &vektor[n] );
```

oder er kann direkt in einen Vergleich eingehen:

```
n = 0;
while( fscanf( eingabe, "%lf", &vektor[n] ) != EOF )
  n = n+1;
```

In der letzten Form wird der Rückgabewert sofort gegen den in `stdio.h` als `define`-Makro definierten Wert `EOF` verglichen, der genau dann zurückgegeben wird, wenn das Dateiende erreicht ist. In der `while`-Bedingung finden also in jedem Durchlauf das Einlesen und die Überprüfung auf das Dateiende statt. `EOF` steht für **E**nd **O**f **F**ile.

2. Operatoren

2.1. Namen

Zur Benennung eigener Variablen, Funktionen und Datentypen sind vom Programmierer Namen zu wählen, deren Aufbau bestimmten Regeln genügen muß. Zulässige Namensbestandteile sind

- Klein- und Großbuchstaben : `a - z` , `A - Z` (keine Umlaute, kein ß)
- Ziffer : `0 - 9`
- Unterstrich : `_`

In Kommentaren dürfen Umlaute und das ß verwendet werden, da Kommentare für den eigentlichen Programmablauf keine Bedeutung haben. Für den Aufbau eines gültigen Namens gelten folgende Regeln:

- Ein Name darf mit einem Kleinbuchstaben, einem Großbuchstaben oder mit einem Unterstrich beginnen, jedoch nicht mit einer Ziffer.
- Groß-/Kleinschreibung wird unterschieden.

Die ersten 31 Zeichen eines Namens werden gemäß Norm mindestens unterschieden. Zusätzlich haben sich folgende Konventionen eingebürgert, die als Empfehlung anzusehen sind:

- Variablen- und Funktionsnamen bestehen aus Kleinbuchstaben.
- `define`-Konstanten bestehen aus Großbuchstaben
- Unterstriche werden nicht als erstes Zeichen verwendet.
- Man verwende sinntragende Namen.

Die folgenden Beispiele erläutern die Regeln:

`TEMP11` , `Temp11` , `temp11`	3 unterschiedliche Namen (Groß-/Kleinschreibung)
`BildMatrix` , `temp_ofen`	zulässige Namen
`__system__` , `____`	zulässig, aber nicht sinnvoll (Kollision mit Systemvariablen möglich)
`Bild-Matrix` , `30x`	nicht zulässig

2.2. *C*-Schlüsselwörter

In ANSI/ISO-*C* gibt es 32 Schlüsselwörter mit fester Bedeutung (Tabelle 2.1). Diese Schlüsselwörter dürfen nicht für Variablennamen, Funktionsnamen und so weiter verwendet werden. Die Schlüsselwörter lassen sich in Gruppen bestimmter Bedeutung zusammenfassen:

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

Tabelle 2.1.: *C*-Schlüsselwörter

2.3. Ganzzahlige Konstanten

In den einleitenden Programmbeispielen wurden bereits Konstanten in Zuweisungen an Variablen und in einfachen Berechnungen verwendet. In *C* ist die Darstellung oktaler, dezimaler und hexadezimaler Konstanten möglich:

Oktale Konstanten werden mit einer führenden Null dargestellt. Die Ziffern 0-7 dürfen zur Zahlendarstellung verwendet werden:

```
x = 077;
```

weist der Variablen `x` den Oktalwert 77 zu (dezimal 63).

Dezimale Konstanten Wie üblich, zum Beispiel `x = 77;`

Hexadezimale Konstanten werden mit den Zeichen `0x` oder `0X` am Anfang dargestellt. Zu den Dezimalziffern kommen die Ziffern `a` - `f` beziehungsweise `A` - `F` des Hexadezimalsystems hinzu. Die Zuweisung

```
x = 0x77;
```

weist der Variablen `x` den Hexadezimalwert 77 zu (dezimal 119).

2.4. Inkrement- und Dekrementoperatoren

Inkrementieren bedeutet Erhöhen eines Zahlenwertes, Dekrementieren bedeutet dagegen Vermindern eines Zahlenwertes. Insbesondere die Erhöhung und Verminderung um 1 ist ein Elementarbefehl der meisten Rechner und kommt auch bei der Programmierung häufig vor. Die Sprache *C* besitzt aus diesem Grund zwei Operatoren, die die Erhöhung und Verminderung *ganzzahliger* Variablen um den Wert 1 durchführen.

```
i++;        /* Variable i um 1 erhöhen    */
k--;        /* Variable k um 1 vermindern */
```

Die beiden Zeilen stellen zwei vollständige Anweisungen dar, die den beiden folgenden Anweisungen völlig gleichwertig sind:

```
i = i+1;    /* Variable i um 1 erhöhen    */
k = k-1;    /* Variable k um 1 vermindern */
```

Der Vorteil wird erst bei Verwendung langer Variablennamen sichtbar. Die beiden Anweisungen

```
messwertzaehler_temperatur  = messwertzaehler_temperatur + 1;
messwertzaehler_temperatur++;
```

sind gleichwertig, die zweite ist jedoch schneller zu schreiben und zu erfassen. Die beiden Operatoren werden zum Beispiel häufig zur Veränderung der Steuervariable in Schleifen verwendet:

```
for( i=0; i<n; i++ )
{
}
```

Bisher wurden die beiden Operatoren an die Variablen angehängt (Postfix-Schreibweise). Die Operatoren können jedoch auch dem Variablennamen vorausgehen (Präfix-Schreibweise). In der Postfixform geschieht die Veränderung des Variablenwertes *nach dem Gebrauch* der Variablen, in der Präfixform geschieht die Veränderung *vor dem Gebrauch*. Dazu folgendes Beispiel:

```
int a, b, c;
c = 0;
a = ++c;    /* nach der Ausführung:  a = 1 , c = 1 */
b = c++;    /* nach der Ausführung:  b = 1 , c = 2 */
```

In der 3. Zeile wird `c` *vor* dem Gebrauch erhöht und sein Wert dann an `a` zugewiesen. In der 4. Zeile wird der Wert von `c` zunächst an `b` zugewiesen und *nach* diesem Gebrauch erhöht. Dadurch erhalten die Variablen die in den Kommentaren angegebenen Werte.

Es gibt Anwendungen, bei denen die Auswertungsreihenfolge nicht (wie beim letzten Beispiel) eindeutig festgelegt ist:

```
i = 0;
while( i<n )
  y[i]  = x[i++];
```

Hier gibt es keine Sicherheit, daß die jeweilige Adresse des Feldelementes `y[i]` auch vor der Erhöhung der Variablen `i` durchgeführt wird! Die Verwendung derselben Variablen mit und ohne Inkrementierung (Dekrementierung) in derselben Anweisung ist also stets zu vermeiden. Die gleichwertige Darstellung mit Hilfe einer `for`-Schleife ist dagegen eindeutig:

```
for( i=0; i<n; i++ )
  y[i]  = x[i];
```

2.5. Zuweisungsoperatoren

Die Verbindung von Rechenoperationen und Zuweisungen sind eine weitere Errungenschaft der Sprache ***C***. Die Operatoren `+=`, `-=`, `*=` und `/=` verknüpfen zunächst die links stehende Variable mit dem Wert der rechten Seite und weisen das so entstandene Ergebnis der links stehenden Variablen zu. Die folgenden Ausdrücke sind deshalb gleichwertig:

Kurzschreibweise	**Ersatzschreibweise**
`a += b;`	`a = a + b;`
`a -= c;`	`a = a - c;`
`a *= x;`	`a = a * x;`
`a /= y;`	`a = a / y;`
`a *= u + v + w;`	`a = a * (u + v + w);`

Die rechte Seite eines Zuweisungsoperators wird immer vollständig ausgewertet, bevor sie in die weitere Rechnung eingeht (Klammerausdruck im letzten Beispiel).

Zuweisungsoperatoren verkürzen die Schreibweise bei langen Variablennamen. Sie können aber stets durch die entsprechende Ersatzschreibweise vermieden werden.

2.6. Vorrang und Bindung bei Operatoren

In der Sprache ***C*** stehen eine Reihe von Operatoren zur Verfügung, für die der Vorrang und die Bindung festgelegt sein müssen.

Die Auswertungsreihenfolge (zum Beispiel „Punktrechnung“ vor „Strichrechnung“) bei Ausdrücken mit mehreren Operatoren kann durch zwei Vorgehensweisen bestimmt werden:

Klammerung Eine bestimmte Auswertungsreihenfolge kann grundsätzlich, wie in der Mathematik, durch entsprechende Klammerung erzwungen werden.

Vorrang Wenn die Eindeutigkeit nicht durch die Klammerung gegeben ist, dann wird die Auswertungsreihenfolge durch den Vorrang der Operatoren bestimmt.
In Tabelle 2.2 sind die Vorränge aller ***C***-Operatoren wiedergegeben. Die Operatoren, die in der Tabelle höher stehen, haben Vorrang vor allen darunterstehenden Operatoren.
Operatoren in derselben Gruppe haben gleichen Vorrang. Hier entscheidet die Bindung über die Auswertungsreihenfolge.

Hierzu einige Beispiele. In der Bedingung in der folgenden `if`-Anweisung werden die Berechnungen vor dem Vergleich ausgeführt, da die beiden arithmetischen Operatoren über den Vergleichen stehen:

```
if ( 2*a <= b+10 )
  y = sin(x);
```

In der Zuweisung

```
y *= a + b + 10;
```

muß zunächst die rechte Seite vollständig berechnet werden, da fast alle Operatoren über den Zuweisungsoperatoren stehen. In der Zuweisung

```
y = 1.0/a*b;
```

kommen die beiden gleichrangigen Operatoren für die Division und die Multiplikation vor. Hier wird nun von links nach rechts ausgewertet. Das heißt aber, daß dieser Ausdruck so ausgewertet wird, als sei er wie folgt geklammert:

```
y = (1.0/a)*b;
```

Die Bedeutung ist also, als Bruch geschrieben, $y = \frac{1}{a} \cdot b$! Der Bruch $y = \frac{1}{a \cdot b}$ muß dagegen unbedingt wie folgt geklammert werden:

```
y = 1.0/(a*b);
```

Einige Operatoren haben eine Bindung nach rechts, das heißt sie gehören zu den Ausdrücken, die ihnen unmittelbar nachfolgen. Beispiele hierfür sind die Vorzeichen und die vorangestellte Inkrement- und Dekrementoperatoren. Auch der Zuweisungsoperator hat eine Bindung nach rechts. Mehrfachzuweisungen, wie zum Beispiel

Operator	Bedeutung	Bindung
`()`	Argumentklammern eines Funktionsaufrufes	links
`[]`	Indexklammern eines Feldelementes	links
`->`	Zeigerzugriff auf ein Strukturelement	links
`.`	Zugriff auf ein Strukturelement	links
`++ --`	Inkrement, Dekrement (Postfix)	links
`++ --`	Inkrement, Dekrement (Präfix)	rechts
`!`	logische Verneinung	rechts
`~`	bitweises Komplement	rechts
`+ -`	Vorzeichen	rechts
`sizeof(`Typ`)`	Speichergröße in Byte	rechts
`(`Zieltyp`)`	explizite Typumwandlung	rechts
`*`	Inhaltsoperator	rechts
`&`	Adreßoperator	rechts
`* / %`	Multiplikation, Division, Divisionsrest	links
`+ -`	Addition, Subtraktion	links
`<< >>`	bitweise nach links, rechts schieben	links
`< <= > >=`	arithmetische Vergleiche	links
`== !=`	arithmetische Vergleiche	links
`&`	bitweise UND	links
`^`	bitweise exklusives ODER	links
`|`	bitweise ODER	links
`&&`	logisches UND	links
`||`	logisches ODER	links
`? :`	bedingter Ausdruck	rechts
`=`	einfache Zuweisung	rechts
`+= -= *= /= %=`	Zuweisungsoperatoren	rechts
`&= ^= |=`		rechts
`<<= >>=`		rechts
`,`	Kommaoperator	links

Tabelle 2.2.: Vorrang und Bindung der *C*-Operatoren

```
x = y = z = 23.9;
```

werden deshalb (sinnvollerweise) von rechts her abgearbeitet, das heißt die Zeile entspricht den Anweisungen

```
z = 23.9;
y = z;
x = y;
```

In solchen Fällen legt die Bindung die Auswertungsreihenfolge fest.

3. Basisdatentypen und Wertebereiche

3.1. Ganzzahlige Datentypen

Es gibt insgesamt vier ganzzahlige Basisdatentypen (`char`, `short`, `int`, `long`), die neben der Grundform jeweils mit den Zusätzen `unsigned` (vorzeichenlos) und `signed` (vorzeichenbehaftet) verwendet werden können.
Größen, die mit der Grundform des Datentyps vereinbart wurden (zum Beispiel `int x;`) sind stets vorzeichenbehaftet. Der Zusatz `signed` ist deshalb überflüssig und wird nur selten verwendet. Vorzeichenlose Größen können nur Beträge als Wert enthalten und daher keine negativen Werte aufnehmen.
Die Bezeichnungen `short` und `short int`, sowie `long` und `long int` sind jeweils gleichbedeutend.

Basistyp	vorzeichenbehaftet	vorzeichenlos
`char`	`signed char`	`unsigned char`
`short` [`int`]	`signed short` [`int`]	`unsigned short` [`int`]
`int`	`signed int`	`unsigned int`
`long` [`int`]	`signed long` [`int`]	`unsigned long` [`int`]
`long long` [`int`]	`signed long long` [`int`]	`unsigned long long` [`int`]

Tabelle 3.1.: Ganzzahlige Basisdatentypen (die Angabe `int` ist bei `short`, `long` und `long long` optional)

3.1.1. Der Datentyp `char`

Der Datentyp `char` zählt (trotz seines Namens: engl. ***char****acter* = Zeichen) zu den ganzzahligen Datentypen. Eine `char`-Größe wird intern in einem Byte (zu 8 Bits) dargestellt. Damit ist grundsätzlich die Darstellung von 2^8 Zahlenwerten möglich.
Zur internen Darstellung des Vorzeichens wird ein Bit verwendet (Bit 7 in Abbildung 3.1), die

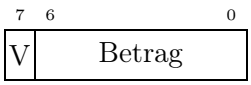


Abbildung 3.1.: Interne Darstellung einer vorzeichenbehafteten `char`-Größe

restlichen 7 Bits nehmen den Betrag auf. Der Wert 0 im Vorzeichen-Bit steht für das positive Vorzeichen. Mit 7 Bits lassen sich die Beträge 0 bis $127 = 2^7$ darstellen. Für den Wert Null wären nun zwei interne Darstellungen vorhanden (mit positivem und negativem Vorzeichen). Da das nicht sinnvoll ist, wird die „negative Null“ als Wert -128 interpretiert. Der Zahlenbereich wird dadurch unsymmetrisch.

Für vorzeichenlose `char`-Werte stehen die 8 Bits zur Darstellung des Betrages zur Verfügung. In diesem Falle umfaßt der Wertebereich die Zahlen $0 \ldots 255 = (2^8 - 1)$.

Datentyp	Zahlenbereich
`signed char`	$-128 \ldots +127$
`unsigned char`	$0 \ldots +255$

Variablen von Typ `char` werden üblicherweise dazu verwendet, einzelne Zeichen eines bestimmten Codes (zum Beispiel des ASCII-Codes) aufzunehmen und bilden damit die Grundeinheit für die Handhabung von Zeichen, Texten und Zeichenketten. Die folgenden Zeilen zeigen die Möglichkeiten:

```
char  zeichen;
zeichen = 'a';          /* ASCII-Wert des Buchstabens a zuweisen     */
zeichen = '\n';         /* ASCII-Wert des Steuerzeichens \n zuweisen */
zeichen = 53;           /* Dezimalwert 53 zuweisen                   */
zeichen = 0x20;         /* Hexadezimalwert 20 zuweisen               */

zeichen = 'a' + 13;     /* ergibt den Wert der Konstanten 'n'        */
```

Zeichenkonstanten sind in einfache senkrechte Anführungsstriche zu setzen. Für Steuerzeichen ist die Ersatzdarstellung mit dem Fluchtsymbol zu verwenden. Zahlenwerte können in der üblichen Form zugewiesen werden.

`char`-Größen können in Berechnungen und Vergleichen verwendet werden. Wegen des sehr eingeschränkten Wertebereiches ist stets auf die Möglichkeit des Zahlenbereichsüberlaufes beziehungsweise -unterlaufes zu achten!

3.1.2. Der Datentyp `int`

Der Basistyp `int` kommt in den Ausprägungen kurze, normale und lange Ganzzahl vor (`short`, `int`, `long`). Die tatsächliche Anzahl der zur internen Darstellung verwendeten Bytes, und damit der Umfang der Zahlenbereiche, hängt vom Rechner und dem verwendeten Compiler ab. Auf Rechnern mit 32 Bit Wortbreite sind folgende Darstellungen üblich:

`short int`	2 Bytes (= 16 Bits)
`int`	4 Bytes (= 32 Bits)
`long int`	4 Bytes (= 32 Bits)

Auf anderen Plattformen (Microcontroller, Großrechner) sind andere Darstellungen zu erwarten. Im Einzelfall muß die Dokumentation eingesehen werden.
Abbildung 3.2 zeigt die interne Darstellung einer `int`-Größe (4-Byte). Das höchstwertige Bit (Nr. 31) enthält das Vorzeichen (0 = positiv). Für den Betrag stehen 31 Bits und damit ein Wertebereich von 0 bis 2147483647 = $(2^{31} - 1)$ zur Verfügung.

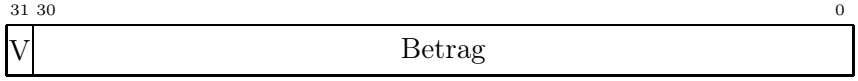


Abbildung 3.2.: Interne Darstellung einer vorzeichenbehafteten `int`-Größe (4 Byte)

Für vorzeichenlose `int`-Werte stehen die 32 Bits zur Darstellung des Betrages zur Verfügung. In diesem Falle umfaßt der Wertebereich die Zahlen $0 \ldots 4294967295 = (2^{32} - 1)$. Da die „negative Null“ wie bei den `char`-Größen den kleinsten negativen Wert darstellt, sind die Wertebereiche der vorzeichenbehafteten `int`-Größen unsymmetrisch.

Datentyp	Bytes	Zahlenbereich
`short`	2	$-32768 \ldots +32767$
`unsigned short`	2	$0 \ldots +65535$
`int`	4	$-2147483648 \ldots +2147483647$
`unsigned int`	4	$0 \ldots +4294967295$
`long`	4	$-2147483648 \ldots +2147483647$
`unsigned long`	4	$0 \ldots +4294967295$

Die Beachtung der Zahlenbereichsgrenzen ist stets die Aufgabe des Programmierers! Die Codezeilen

```
int x = 100000, y;

y = x*x/x;

printf ("y = %d\n", y );
```

stellen die Berechnung des Bruches $y = \frac{x \cdot x}{x}$ dar. Die Variable `x` hat den nicht ungewöhnlichen Wert 10^5 und man erwartet auch diesen Wert als Ergebnis in der Variablen `y`. Der durch die `printf`-Anweisung tatsächlich ausgegebene Wert ist jedoch 14100!
Die Multiplikation `x*x` im Zähler ergibt das Zwischenergebnis 10^{10}, welches bereits außerhalb des Zahlenbereiches einer `int`-Größe liegt. Hier hat ein Zahlenbereichsüberlauf stattgefunden. Das Programm rechnet ohne Warnung oder Abbruch mit dem falschen Zwischenergebnis weiter.

3.2. Reelle Datentypen

Es gibt drei reelle Datentypen (`float`, `double`, `long double`), die sich nur hinsichtlich ihrer Zahlenbereiche und der erreichbaren Darstellungsgenauigkeit der Zahlen unterscheiden. In Tabelle 3.2 ist eine Übersicht angegeben.

Datentyp	**Byte**	**Bit**	**Zahlenbereich**	**Genauigkeit** (gültige Stellen)
`float`	4	32	$\approx -10^{38} \ldots \approx +10^{38}$	$\sim$7
`double`	8	64	$\approx -10^{308} \ldots \approx +10^{308}$	$\sim$15
`long double`	10	80	$\approx -10^{4932} \ldots \approx +10^{4932}$	$\sim$19

Tabelle 3.2.: Reelle Datentypen

Wie bei den Ganzzahlen, lassen sich die Zahlenbereiche und die Darstellungsgenauigkeiten aus der internen Zahlendarstellung ableiten. Dies soll hier an einem Beispiel für den Datentyp `float` erläutert werden.

Die reelle Zahl 5,375 läßt sich ohne Mühe in eine Summe von Zweierpotenzen entwickeln:

$$5,375 \quad = \quad 4+1+\frac{1}{4}+\frac{1}{8} \quad = \quad 2^2+2^0+2^{-2}+2^{-3}$$

Daraus ergibt sich die Binärdarstellung

$$101,011$$

die, bei Erhaltung des Wertes, immer so umgeformt werden kann, daß eine 1 vor dem (Binär-) Komma steht:

$$+1,01011 \cdot 2^2$$

Aus dieser sogenannten normalisierten Darstellung lassen sich nun drei Bestandteile entnehmen, die als Bitmuster in einer internen Zahlendarstellung abgespeichert werden können:

$$\underbrace{+}_{Vorzeichen} \quad 1,\ \underbrace{01011}_{Mantisse} \cdot 2 \overbrace{2}^{Exponent}$$

Vorzeichen Ein Bit (0 = positiv).

Mantisse Die Nachkommastellen der Binärdarstellung. Die Eins vor dem Komma wird nicht gespeichert, da sie (außer bei dem Wert 0.0) wegen der Normalisierung immer vorhanden ist.

Exponent Da die interne Darstellung immer binär ist, reicht die Abspeicherung des (binären) Exponenten. Da der Exponent auch negativ sein kann, wird vor der Abspeicherung eine feste Verschiebung addiert. Damit erscheinen in der endgültigen internen Darstellung nur positive Werte im Exponentenanteil.

Für `float`-Größen sind die Verhältnisse in Abbildung 3.3 dargestellt. Im höchstwertigen Bit ist das Vorzeichen gespeichert. Für den Exponenten werden 8 Bits verwendet, für die Mantisse 23 Bits.

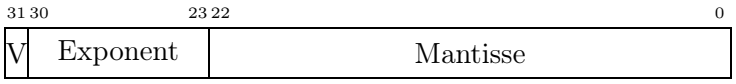


Abbildung 3.3.: Interne Darstellung einer `float`-Größe (4 Byte)

Der Beispielwert 5,375 hat die interne Darstellung, die in Abbildung 3.4 wiedergegeben ist. Zum Exponenten 2 wurde die Verschiebung 127 (binär 1111111) addiert.

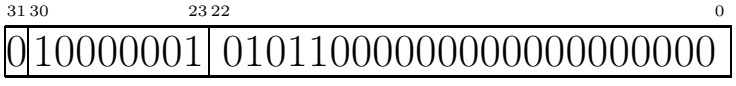


Abbildung 3.4.: Interne Darstellung des Wertes 5,375 als `float`-Größe

Aus der Kenntnis der internen Darstellung läßt sich auch der Zahlenbereichsumfang abschätzen. Der größte Betrag, der mit einer Mantisse darstellbar ist, ist annähernd 1. Wegen der nicht gespeicherten, aber bei der Rechnung stets berücksichtigten Eins vor dem Dualkomma ist der größte Grundwert also dual 1,11111111111111 , das heißt annähernd 2. Der größte darstellbare Exponent ist +127. Damit ergibt sich die größte darstellbare Zahl zu $2 \cdot 2^{+128} \approx 3,4 \cdot 10^{+38}$

(siehe Tabelle 3.2).
Für die Nachkommastellen (Mantisse) stehen 23 Bits zur Verfügung. Damit sind Zahlen in der Größenordnung $2^{23} \approx 8 \cdot 10^6 \approx 7$ Dezimalstellen darstellbar. Die Darstellungsgenauigkeit ist beim Datentyp `float` grundsätzlich auf 7 Dezimalstellen begrenzt (siehe Tabelle 3.2).

Für die Datentypen `double` und `long double` können die gleichen Abschätzungen vorgenommen werden.

Zur Darstellung reeller Zahlen sollte, wenn nicht ausdrücklich anders gefordert, der Typ `double` verwendet werden.

Die Beachtung von Zahlenbereichsgrenzen und Darstellungsgenauigkeiten ist natürlich dem Programmierer überlassen. Nach dem oben gesagten macht es wenig Sinn, eine Konstante mit 200 Stellen an eine `double`-Variablen zuzuweisen:

```
mein_pi = 3.14159265358979323846264338327950288 ... ;
```

Ebensowenig macht es Sinn, ein Berechnungsergebnis auf 50 Stellen auszugeben, in der Hoffnung, daß die letzten Stellen noch irgendeine sinnvolle Information enthalten.

3.3. Darstellungsbedingte Rundungsfehler

Eine Folge der internen Zahlendarstellung sind Rundungsfehler. Diese entstehen in jedem Zahlensystem, wenn für die Darstellung bestimmter Zahlen nur eine begrenzte Anzahl Stellen zur Verfügung steht.

Im Dezimalsystem ist der Bruch 1/10 als reelle Zahl $0,1$ ohne Rundungsfehler darstellbar, während der Bruch 1/3 als reelle Zahl $0,333333\ldots$ nach endlich vielen Stellen mit Rundungsfehler abgebrochen werden muß.

Im Binärsystem verhält sich das ebenso, allerdings nicht unbedingt bei den gleichen Zahlenwerten. Im Binärsystem ist der Bruch 1/10 nicht exakt darstellbar, weil er eine unendliche Dualentwicklung besitzt:

$$\frac{1}{10} = 0,0\overline{0011}_2$$

Die Darstellung dieses Wertes ist wegen der begrenzten Mantissenlänge zwangsläufig abgeschnitten, der Wert kann intern nur mit einem kleinen Rundungsfehler dargestellt werden.

Diese Erkenntnis hat mehrere Folgen für die Verwendung von Zahlen in Programmen. Eine der wichtigsten ist die, daß reelle Größen nicht direkt in Bedingungen auf Gleichheit geprüft werden dürfen.

In den folgenden Zeilen wird durch Berechnung der Variablen `x` ein Wert zugewiesen und anschließend dieser gegen den Wert $1,75$ verglichen:

```
x = .... ;
if( x == 1.75 )
{
}
```

Selbst wenn theoretisch der Wert $1,75$ genau berechnet werden muß, kann sich auf Grund von Rundungsfehlern bei jeder Berechnung eine geringfügige, praktisch unbedeutende Abweichung ergeben, so daß die Bedingung selten oder nie wahr ist ! Wenn die Berechnung wiederum von Eingabewerten oder sich ändernden Berechnungsparametern abhängt, dann wird das Ergebnis der Auswertung der Bedingung zum Zufallsereignis !

Ein Ausweg ist die Einrichtung eines Toleranzwertes, zum Beispiel als `define`-Größe `EPSILON`. Wenn der *Betrag* des Berechnungsergebnisses höchstens um diesen Toleranzwert vom Vergleichswert $1,75$ abweicht, dann ist die Bedingung wahr:

```
#define  EPSILON  1.0E-10

x = .... ;
if( fabs(x -1.75) <= EPSILON )
{
}
```

Eine direkte Folge davon ist, daß sich Schleifensteuerungen mit reellen Zahlen von selbst verbieten. Ob und wann der Wert der Variablen `x` in den nachfolgenden Zeilen mit dem eingelesenen Endwert `endwert` übereinstimmt ist nicht vorhersehbar. Ob der letzte Durchlauf der Schleife stattfindet wird hier zum Zufallsereignis:

```
double  inkrement, endwert, x;

inkrement = 0.1;
endwert   = scanf( "%lf", &endwert );

for ( x=0.0; x<=endwert; x=x+inkrement )
{
}
```

Da die Anzahl von Schleifendurchläufen der Natur nach ganzzahlig ist, kommen zur Schleifensteuerung grundsätzlich nur *ganzzahlige Steuervariablen* in Betracht. In den folgenden Zeilen werden die Anzahl der Durchläufe `n` und die Schrittweite `inkrement` berechnet. Der Wert der Variablen `x` wird in jedem Durchlauf aus dem aktuellen Wert des Schleifenzählers ermittelt. Die Anzahl der Durchläufe ist damit immer gleich.

```
int     i, n;
double  x, inkrement, endwert;

n         = 250;          /* Anzahl Schleifendurchläufe */
endwert   = 30.0;         /* Endwert                    */
inkrement = endwert/n;

for ( i=0; i<=n; i++ )
{
  x = i*inkrement;
  ...
}
```

3.4. Typumwandlung

In ***C*** sind Ausdrücke mit gemischten Datentypen erlaubt und häufig auch erforderlich. Für die Auswertung solcher Ausdrücke gelten einige Regeln, die in diesem Abschnitt vorgestellt werden sollen.

3.4.1. Implizite Typumwandlung

Wenn keine expliziten Umwandlungsregeln angegeben sind (Behandlung im nächsten Abschnitt), dann werden unter Umständen trotzdem implizite oder automatische Umwandlungen

durchgeführt. Es gelten zunächst zwei Grundregeln:

1. Wenn zwei Operanden mit demselben Datentyp verknüpft werden, dann besitzt das (Zwischen-) Ergebnis zunächst ebenfalls diesen Datentyp.
2. Wenn zwei Operanden mit unterschiedlichen Datentypen verknüpft werden, dann besitzt das (Zwischen-) Ergebnis in den meisten Fällen den in Bezug auf den Zahlenbereich weiteren Datentyp.

Hierzu einige Beispiele. Zunächst sollen Zuweisungen betrachtet werden:

```
int     i;
float   f;
double  d;
  ...
i = f;       /* evtl. Informationsverlust durch Abschneiden         */
i = d;       /*  von Nachkommastellen                               */
f = d;
  ...
f = i;       /* float-Wert kann vom int-Wert abweichen              */
  ...
d = i;       /* kein Informationsverlust; die Variablen            */
d = f;       /*  auf der linken Seite haben die weiteren Datentypen */
```

Ganzzahlen haben keine Nachkommastellen, `float`-Größen haben weniger Nachkommastellen als `double`-Größen. Die ersten drei Zuweisungen können, müssen aber nicht, zu Informationsverlust führen. Bei den zwei letzten Zuweisungen wird stets an weitere Formate zugewiesen. Hier kann kein Informationsverlust auftreten.
Bei der Zuweisung eines `int`-Wertes an eine `float`-Größe (beide in 32 Bit-Darstellung, siehe Seite 40 und Seite 42) ist folgendes zu beachten. Für die Darstellung des `int`-Betrages stehen 32 Bits zur Verfügung. Für die Darstellung des `float`-Wertes sind dagegen nur 23 Bits in der Mantisse vorhanden. Bei Werten mit großen Beträgen werden die letzten Bits der Ganzzahldarstellung nicht in die Mantisse übernommen. Wird zum Beispiel der Wert 100000059 an eine `float`-Variable zugewiesen, dann hat diese anschließend den Wert 100000056 !

In den folgenden Zeilen wird die `int`-Variable `a` mit der reellen Konstanten `0.5` multipliziert. Zunächst gilt Regel 2: das Zwischenergebnis für die rechte Seite ist (intern) der reelle Wert `3.5`.

```
int a = 7;

a = 0.5*a;      /* a erhält den Wert 3 */
```

Da die Zuweisung dieses Zwischenergebnisses an eine `int`-Variable erfolgt, wird der nur intern vorhandene reelle Wert automatisch in eine `int`-Größe verwandelt. Dabei geht natürlich die Nachkommastelle des Ergebnisses verloren. Die Variable `a` besitzt so den Wert 3.

Im folgenden Beispiel soll die Variable `c` mit dem Faktor 1/2 multipliziert werden. Das Ergebnis ist aber überraschenderweise nicht 4 sondern 0 !

```
double  c = 8.0;

c *= 1/2;       /* c erhält den Wert 0 ! */
```

Auf der rechten Seite werden zunächst zwei *ganzzahlige* Konstanten dividiert. Nach Regel 1 ist das Ergebnis ebenfalls ganzzahlig und damit Null. Die anschließende Multiplikation mit `c`

setzt diese Variable ebenfalls zu Null. Die verwendete Schreibweise `1/2` für den Faktor ist zwar eingängig aber falsch. Mit der reellen Darstellung `0.5` wären dagegen keine Schwierigkeiten aufgetreten.

Die Regeln für die implizite Typumwandlung gelten auch für die Rückgabewerte und die Parameterübergabe von Funktionen. Die folgende Funktion erhält ein `int`-Argument und gibt einen `int`-Wert zurück:

```
int quad ( int x )
{
  return x*x;
}
```

Beim Aufruf wird ein `double`-Wert übergeben und das Funktionsergebnis an eine `double`-Variable zugewiesen.

```
double  a, b = 2.9;

a = quad( b );   /* a erhält den Wert 4 */
```

Da das Funktionsargument vom Typ `int` ist, wird der Wert `2.9` bei der Übergabe in den `int`-Wert `2` gewandelt. Zurückgegeben wird dann der `int`-Wert `4`. Dieser wird bei der Rückgabe automatisch in eine `double`-Größe) gewandelt.

Die beschriebenen Auswirkungen der automatischen Typumwandlung können gewünscht sein und damit gezielt eingesetzt werden. Oft sind aber Unkenntnis oder Mißachtung dieser Umwandlungsregeln die Ursache schwer aufzufindender Fehler.
Das ist zum Beispiel dann der Fall, wenn der fehlerhafte Ausdruck nur ein Summand einer Summe ist, in der er nur einen kleinen Teil zum Gesamtergebnis beiträgt oder möglicherweise gar nicht immer auftritt. Der letzte Summand in der nachfolgenden Summe ist, unabhängig vom Wert der Variablen `x`, wegen des Faktors `(1/8)` immer Null:

```
y = 4.1*( sin(x) + 28*cos(3*x) ) - 3.7*( 2*x - 13*x*x ) + (1/8)*exp(x);
```

3.4.2. Explizite Typumwandlung

Eine Typumwandlung kann ausdrücklich durch die Verwendung eines Umwandlungsoperators erzwungen werden. Der Umwandlungsoperator besteht aus dem Namen des Zieldatentyps, der in runden Klammern vor den umzuwandelnden Ausdruck gestellt wird.
Das folgende Beispiel zeigt die Verwendung für eine Umwandlung vom Typ `int` nach `double`:

```
int     i;
double  y;
  ...
y = (double)i;      /* explizite Typumwandlung */
```

Die Verwendung dient hier nur zur Dokumentation. Die entsprechende Umwandlung würde ohne den Operator als implizite Umwandlung genau so stattfinden.
Die nächsten Zeilen enthalten eine implizite und eine explizite Typumwandlung:

```
double  d = 7.8;

d = (int)(d+0.5);   /* echte Rundung */
```

Der Wert der Variablen `d` wird um `0.5` erhöht und dann explizit in eine `int`-Größe (Wert 8) gewandelt. Dieses Zwischenergebnis wird durch implizite Wandlung in einen `double`-Wert gewandelt und wieder `d` zugewiesen. Der Wert von `d` wurde dadurch echt gerundet (das heißt zur nächsten ganzen Zahl hin).

Die wichtigsten Anwendungen der expliziten Typumwandlungen liegen bei der Umwandlung von Zeigern.

4. Ablaufsteuerung

In diesem Kapitel werden zunächst Vergleichsoperatoren und logische Operatoren vorgestellt. Anschließend werden weitere Konstrukte zur Ablaufsteuerung eingeführt. Die `while`- und die `for`-Schleife wurden bereits in Abschnitt 1.7 beschrieben und werden deshalb hier nicht noch einmal aufgenommen.

4.1. Vergleichsoperatoren

Für arithmetische Vergleiche stehen folgende Vergleichsoperatoren zur Verfügung.

`<`	kleiner	`<=`	kleiner oder gleich
`>`	größer	`>=`	größer oder gleich
`==`	Gleichheit	`!=`	Ungleichheit

Das Ergebnis eines Vergleiches ist stets eine Wahrheitswert. Da in ***C*** (wohl aber in ***C++***) kein eigener Datentyp für Wahrheitswerte vorhanden ist, werden diese intern durch ganze Zahlen dargestellt:

`0`	$\hat{=}$	falsch
`1` bzw. $\neq$ `0`	$\hat{=}$	wahr

Da bereits einige Anwendungsbeispiele gegeben wurden, soll hier nur auf auf einen gelegentlich auftretenden Fehler hingewiesen werden. In den folgenden Zeilen wird `a` zu `0` gesetzt und bis zur `if`-Anweisung nicht mehr verändert. Die Bedingung lautet hier `a = 1`. In den meisten Fällen ist das ein Schreibfehler — gemeint war `a == 1` :

```
int a = 0;
  ...
if( a = 1 )     /* Operator "==" ? */
{
}
```

Der Ausdruck `a = 1` ist fehlerfreies ***C***, stellt aber eine *Zuweisung* dar, die an dieser Stelle tatsächlich erlaubt ist. Die Variable `a` erhält hier den Wert 1 und dieser Wert wird dann auch als Wahrheitswert verwendet und bedeutet *wahr*. Damit wird sowohl der Variablenwert verändert als auch ungewollt die bedingten Anweisung im `if`-Block ausgeführt.
Derartige Fehler sind auf Grund des ähnlichen Schriftbildes schwer zu finden. Hier hilft nur Sorgfalt und die Beachtung von Compiler-Warnungen.

4.2. Logische Operatoren

Zur Bildung zusammengesetzter Bedingungen sind die in der zweistelligen Logik üblichen Operatoren UND, ODER und Verneinung erforderlich. Die zugehörigen Operatoren werden wie folgt geschrieben:

`&&`	logisches UND
`||`	logisches ODER
`!`	logische Verneinung

Hier einige Beispiele; es gelten die in der Logik üblichen Verknüpfungsregeln:

```
int i=1, j=2, k=3;

 (   i ==  1 ) && ( k <   4 )   /* wahr   */
 (   i  <  3 ) && ( k > 100 )   /* falsch */
 ( j*k  < 10 ) || ( k > 100 )   /* wahr   */
!(   1  <  j )                  /* falsch */
```

Die Auswertung logischer Ausdrücke wird üblicherweise abgebrochen, sobald das Ergebnis feststeht (abgekürzte Auswertung, engl. *short circuit evaluation*). Wenn in einer reinen UND-Verknüpfung ein Term gefunden wird der falsch ist, dann kann die Auswertung abgebrochen werden, weil das Gesamtergebnis nicht mehr richtig werden kann. Bei einer reinen ODER-Verknüpfung legt entsprechend der erste wahre Term das Gesamtergebnis fest.

Die abgekürzte Auswertung braucht weiter nicht beachtet zu werden, außer in den Fällen, in denen in den nicht mehr ausgewerteten Termen Funktionsaufrufe stehen. Diese werden dann natürlich ebenfalls nicht mehr ausgeführt (ein sogenannter Nebeneffekt).

```
int cnt = 0;
  ...
while( ( ++cnt >= 3 ) && ( c=getchar() ) != EOF )
{
}
```

Wenn in den voranstehenden Zeilen der Wert von `cnt` nach der Anfangswertzuweisung nicht mehr verändert wird, dann ist der Term `(++cnt>=3)` falsch und die Bewertung der gesamten Bedingungen wird abgebrochen. Dadurch wird aber das nächste Zeichen (Aufruf von `getchar()`) nicht mehr eingelesen. Ob das tatsächlich gewünscht ist, muß im Einzelfall entschieden werden.

4.3. `if-else`-Anweisung

Die `if-else`-Anweisung wurde bereits auf Seite 17 beschrieben. Das zugehörige Flußdiagramm ist in Abbildung 4.1 wiedergegeben. Es kann offensichtlich nur der `if`-Teil oder der `else`-Teil durchlaufen werden, jedoch niemals beide Teile. Wenn der `else`-Teil fehlt, spricht man von einer bedingten Anweisung.

Durch Schachtelung von `if-else`-Anweisungen ist die Darstellung von binären Entscheidungsbäumen möglich: In Liste 4.1 ist ein solche Schachtelung mit insgesamt 8 Fällen wiedergegeben. Diese Programmstrukturen werden allerdings bereits bei geringen Schachtelungstiefen unübersichtlich und sollten deshalb durch andere Konstruktionen (Auflösung in Funktionen, `switch`-Anweisungen) ersetzt werden.

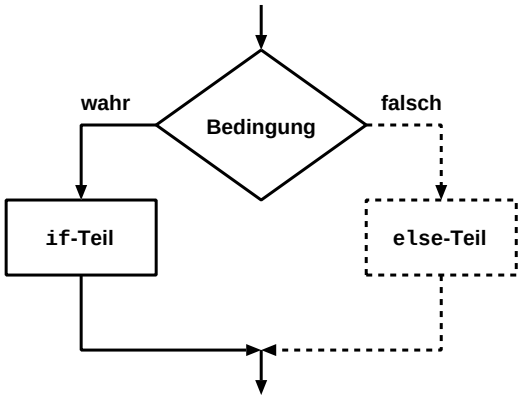


Abbildung 4.1.: `if-else`-Anweisung

Liste 4.1: Binärbaum durch Schachtelung von `if-else`-Anweisungen

```
1  if (a!=0) {
2    if (b!=0) {
3      if (c!=0) {       /* ====   a &&  b &&  c === */
4      }
5      else {            /* ====   a &&  b && !c === */
6      }
7    }
8    else {
9      if (c!=0) {       /* ====   a && !b &&  c === */
10     }
11     else {            /* ====   a && !b && !c === */
12     }
13   }
14 }
15 else {
16   if (b!=0) {
17     if (c!=0) {       /* ====  !a &&  b &&  c === */
18     }
19     else {            /* ====  !a &&  b && !c === */
20     }
21   }
22   else {
23     if (c!=0) {       /* ====  !a && !b &&  c === */
24     }
25     else {            /* ====  !a && !b && !c === */
26     }
27   }
28 }
```

Eine wichtige Regel bei der Verwendung von `if-else`-Anweisungen regelt die Zusammengehörigkeit der einzelnen Teile bei geschachtelten Anweisungen:

Ein `else` gehört immer zu dem *vorausgehenden* `if`.

Hierzu folgendes Beispiel:

```
x = -2;
y = +3;
if( x < 0 )
  if( y < 0 )  printf("Beide Werte negativ!");
else
  printf("x >= 0 !");
```

Obwohl `x` kleiner Null ist, lautet die Ausgabe

```
x >= 0 !
```

Das `else` gehört (trotz der Einrückung!) zu der unmittelbar davor stehenden Zeile `if( y < 0 ) ...` und nicht zum ersten `if`!

4.4. `do-while`-Schleife

Die allgemeine Form der `do-while`-Schleife lautet:

```
do
{
  Schleifenkörper
} while( Bedingung );
```

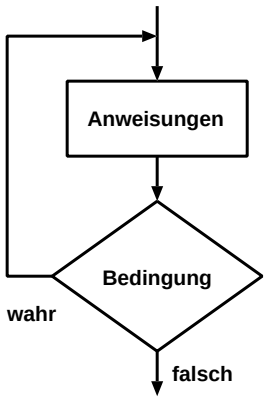


Abbildung 4.2.: `do-while`-Schleife

Der wesentliche Unterschied zur `while`-Schleife besteht darin, daß die Abbruchbedingung jeweils *nach* jedem Durchlauf geprüft wird. Damit ist gewährleistet, daß mindestens ein Durchlauf stattfindet.

Die `do-while`-Schleife kann damit immer dann eingesetzt werden, wenn die Abbruchbedingung durch den Inhalt des Schleifenkörpers beeinflußt wird (zum Beispiel durch Eingaben oder Berechnungen).

Das folgende Beispiel zeigt die Eingabe eines Prozentsatzes, der zwischen den Werte 0 bis 150 liegen darf. Die Durchführung einer weiteren Eingabe wird solange erzwungen, bis der eingegebene Wert in den erlaubten Grenzen liegt:

```
do
{
  printf( "\nProzentsatz (0...150) = " );
  scanf ( "%d", &prozent );
}
while( prozent<0 || prozent>150 );
```

Die Durchführung mehrerer Berechnungen in einer Schleife läßt sich wie folgt umsetzen:

```
do
{
  ...
  printf("\n... weiter (j/n) : ");
}
while( getchar() != 'n' );
```

Solange nach der Abfrage am Schleifenende nicht genau das Zeichen `n` eingegeben wird, ist die Bedingung falsch und die Schleife wird erneut ausgeführt.

4.5. `switch`-Anweisung

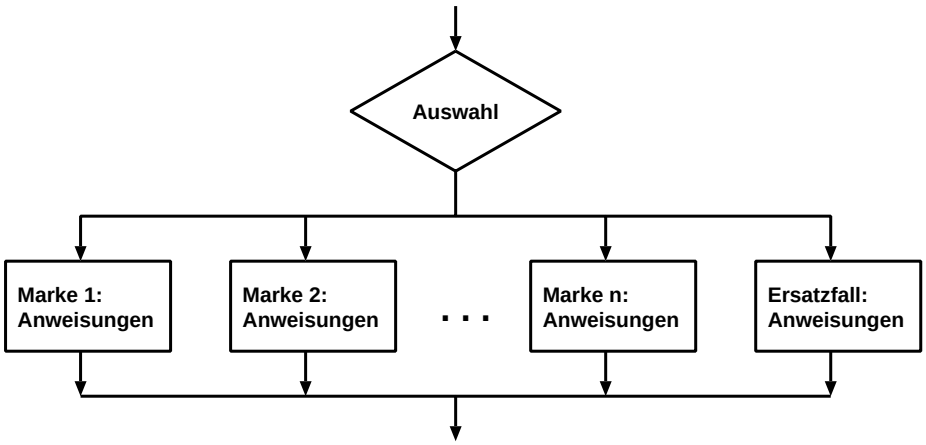


Abbildung 4.3.: `switch`-Anweisung

Die `switch`-Anweisung dient zur wahlweisen Ausführung linear angeordneter Fälle (Abbildung 4.3). Die Auswahl eines Falles geschieht am Anfang der Konstruktion durch die Auswertung einer Auswahlanweisung. Diese Auswahlanweisung muß ein ganzzahliges Ergebnis liefern. Die unterschiedlichen Fälle sind jeweils mit Fallmarken versehen, die zur Übersetzungszeit einen konstanten Wert ergeben müssen (siehe unten).
Bei der Programmausführung wird der Fall angesprungen, für den der ermittelte Wert des Auswahlausdruckes und die Fallmarke übereinstimmen. Wenn kein Fall mit dieser Marke vorhanden ist wird der letzte Fall mit der Marke `default` angesprungen. Der sogenannte `default`-Fall darf fehlen. Dann wird das Programm nach der `switch`-Anweisung fortgesetzt.

Die allgemeine Form der `switch`-Anweisung lautet:

```
switch( Auswahlausdruck )
{
  case konst-Ausdruck-1:
    Anweisungen
    break;

  case konst-Ausdruck-2:
    Anweisungen
    break;

  default:
    Anweisungen
}
```

Jeder Fall wird in der Regel durch die Anweisung `break` beendet. `break` ist ein unbedingter Sprung der die gesamte `switch`-Anweisung verläßt. Die `break`-Anweisung darf allerdings auch fehlen. Dann werden die Anweisungen des nächsten Falles ausgeführt (falls vorhanden), das heißt die Programmausführung fällt in den nächsten Fall hinein (engl. *fall through*). Das kann beabsichtigt sein, führt aber dann zu Fehlern, wenn das `break` einfach vergessen wurde.

Die Fallmarken müssen ganzzahlig und zur Übersetzungszeit konstant sein, das heißt sie dürfen aus beliebigen ganzzahligen Konstanten bestehen, oder aus Rechenausdrücken, die sich zu einer Konstanten bewerten lassen. Folgende Marken sind zum Beispiel möglich:

```
case  0xFF:        /* hexadezimale Konstante */

case  255:         /* dezimale Konstante     */

case  055:         /* oktale Konstante       */

case  'n':         /* Zeichenkonstante       */

case  100+29:      /* Berechnung             */

case  'A'+9:       /* Berechnung             */

case  WORT:        /* define-Makro : 1       */
```

Die Werte der Fallmarken müssen keineswegs numerisch lückenlos aufeinanderfolgen und sie müssen auch nicht geordnet sein.

Das folgende Beispiel zeigt die Verwendung einer `switch`-Anweisung im einfachste Fall. Die Fälle sind von 1 bis 6 durchnummeriert, der Auswahlausdruck ist eine einfache `int`-Variable.

```
int note;
  ...
switch(note)
{
case 1:
        printf ("Note: Sehr gut");
        break;
case 2:
```

```
        printf ("Note: Gut");
        break;
case 3:
        printf ("Note: Befriedigend");
        break;
case 4:
        printf ("Note: Ausreichend");
        break;
case 5:
        printf ("Note: Mangelhaft");
        break;
case 6:
        printf ("Note: Ungenügend");
        break;
default:
        printf ("Notenwert existiert nicht!");
}
```

4.6. break-Anweisung

Die break-Anweisung wurde im letzten Abschnitt als Bestandteil der switch-Anweisung eingeführt, sie beendet aber auch Schleifen durch eine unbedingten Sprung hinter die Schleifenanweisung.
Im folgenden Beispiel soll festgestellt werden, ob ein bestimmter Zahlenwert im Feld feld vorhanden ist. Die Lösung könnte wie folgt aussehen:

```
#define  N  1000
  ...
int feld[N], i, n = N, wert;
  ...
for( i=0; i<n; i++ )
  if( feld[i] == wert ) break;

if(i<n)
  printf("\nWert gefunden");
else
  printf("\nWert nicht gefunden");
```

Als Alternative kann die if-Bedingung mit dem Abbruchkriterium verknüpft werden:

```
for( i=0; (i<n) && (feld[i] != wert); i++ )
  ;                                              /* EMPTY */
```

Ob die for-Schleifen vorzeitig durch break oder beim Erreichen des letzten Feldplatzes (Wert nicht im Feld vorhanden) abgebrochen wurde, kann hier nur am Stand des Schleifenzählers i erkannt werden: Wenn dessen Wert kleiner 1000 ist, dann wurde die Schleife vorzeitig abgebrochen. Wenn sein Wert genau 1000 ist, dann ist die Schleife bis zum Ende durchgelaufen.
Die Erhöhung eines Schleifenzählers ist immer die letzte Maßnahme vor dem nächsten Durchlauf. Nach dem letzten vollständigen Durchlauf wird der Zähler von 999 auf 1000 erhöht, danach ist die Abbruchbedingung i<n falsch und das Programm wird hinter der Schleife fortgesetzt.
In den meisten Anwendungsfällen können und sollen unbedingte Sprünge wie break durch zusammengesetzte Bedingungen oder die Verwendung von if-else vermieden werden.

5. Funktionen

Die wichtigste Vorgehensweise bei der Lösung umfangreicher Aufgaben ist deren Zerlegung und Lösung in handhabbare und überschaubare Teilaufgaben. Bei Programmierprojekten gibt es mehrere Gliederungsstufen. Die Zusammenfassung logisch zusammengehöriger Aufgabenteile zu einer abgeschlossenen Einheit, einer Funktion, ist eine davon. Für die Nutzung einer derartigen Funktion ist dann im wesentlichen nur die Kenntnis der Schnittstelle erforderlich (Aufrufparameter und Rückgabewerte).
Eine weitere Gliederungsstufe ist die Zusammenfassung von Funktionen in einzeln übersetzbare Dateien, die möglicherweise von unterschiedlichen Autoren stammen und in Funktionsbibliotheken abgelegt werden können.

5.1. Funktionsdefinition

Die Definition und die Verwendung einfacher Funktionen wurde bereits in Abschnitt 1.8 eingeführt. Wegen der Bedeutung und der hinzukommenden Erweiterungen wird das Wichtigste hier wiederholt.

Die Definition einer ***C***-Funktion besitzt folgenden allgemeinen Aufbau:

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

Es gelten folgende Regeln:

- Der Funktionsname ist frei wählbar (siehe Abschnitt 2.1).
- Ein Rückgabetyp muß angegeben werden.
- Die Parameterliste enthält formale Parameter, das heißt Platzhalter für die beim Aufruf tatsächlich zu übergebenden Werte. Die einzelnen Parameter werden durch Kommata getrennt.
- Für jeden Parameter ist der Datentyp anzugeben.
- Die Parameterliste kann leer sein; die runden Klammern müssen aber in jedem Fall vorhanden sein.
- Vereinbarungen und Anweisungen können beide fehlen.

Die folgenden Zeilen stellen die Definition einer Umrechnungsfunktion dar. Der Datentyp des Rückgabewertes ist `double`. Das Ergebnis wird innerhalb der Funktion in der `double`-Variablen `erg` errechnet und über `return` zurückgegeben. Der Typ dieser Hilfsvariablen paßt also zum Rückgabetyp der Funktion.

```
double  celsius ( double fahrenheit )
{
  double  erg;                                  /* Hilfsvariable        */
  erg = (5.0/9.0)*(fahrenheit-32.0);            /* Umrechnungsformel    */
  return  erg;                                  /* Ergebnis zurückgeben */
}
```

Der (einzige) Funktionsparameter `fahrenheit` ist lediglich ein benannter Platzhalter (ein sogenannter formaler Parameter) für die unabhängige Eingangsgröße der Funktion. Ohne diesen Platzhalter ließe sich die Berechnung nicht formulieren. Ein formaler Parameter entspricht einer unabhängigen Variablen in der Definition einer mathematischen Funktion.

Alleine dadurch, daß eine Funktion definiert ist, findet noch keine Berechnung statt. Das geschieht erst, wenn die einmal definierte Funktion mit tatsächlichen Argumentwerten, den sogenannten aktuellen Parametern, aufgerufen wird:

```
double    fahr, cels;                         /* Vereinbarungen              */
  ...
fahr  = 22.45;
cels  = celsius( fahr );                      /* Aufruf der Funktion celsius */
  ...
```

Der Wert des aktuellen Parameters `fahr` des Aufrufes ersetzt den formalen Parameters `fahrenheit` der Funktionsdefinition. Damit sind alle Größen der rechten Seite des Funktionsaufrufes bestimmt und die Berechnung kann durchgeführt werden. Nach durchgeführtem Aufruf wird der in der Funktion errechnete Wert an die Variable `cels` in der Aufrufzeile zugewiesen.

Der Rückgabetyp `void`

Gelegentlich müssen Funktionen definiert werden, die von ihrer Aufgabenstellung her keinen sinnvollen Rückgabewert haben. Da grundsätzlich ein Rückgabetyp angegeben werden muß, müßte in solchen Fällen ein nutzloser Wert (zum Beispiel 0, Rückgabetyp `int`) zurückgeben werden. Diese Unschönheit kann durch die Verwendung des Datentyps `void` (engl. *void* = nichts, leer) umgangen werden. Die folgende Funktion gibt lediglich einen Meldungstext aus:

```
void meldung_7 ( void )
{
  printf("Eingabedatei nicht vorhanden!");
  return;
}
```

Der Rückgabetyp `void` bedeutet hier, daß die Funktion keinen Rückgabewert besitzt und deshalb ein Funktionsaufruf `meldung_7 ( )` nicht auf der rechten Seite einer Zuweisung stehen kann. Eine Funktion ohne Rückgabewert nennt man auch Prozedur.
Die Parameterliste ist ebenfalls leer. Hier würde ein leeres Klammernpaar in der Definition genügen. Die Angabe `void` als Parameterliste dokumentiert hier lediglich, daß es sich um eine leere Liste handelt.

Die `return`-Anweisung

In der Funktion `meldung_7 ( )` gibt es keinen Rückgabewert und deshalb hat die `return`-Anweisung die Form

```
return;
```

Diese Anweisung darf hier auch fehlen.

Bei Funktionen mit Rückgabewert kann nach `return` ein beliebiger Ausdruck stehen, der sich zur Ausführungszeit des Programmes zu einem Wert des geforderten Rückgabetypes bewerten läßt, oder durch automatische Typumwandlung in eine solchen umgewandelt wird. Zulässig sind zum Beispiel

```
return 1;                       /* Konstante         */
return x;                       /* einfache Variable */
return x*x;                     /* Rechenausdruck    */
return sin(2*x);                /* Funktionsaufruf   */
```

Grundsätzlich gilt:

- Eine `return`-Anweisung beendet die Funktionsausführung unmittelbar und veranlaßt einen Rücksprung an die Aufrufstelle. Bei echten Funktionen (nicht bei Prozeduren) wird der Rückgabewert zurückgeben.
- Eine Funktion kann beliebig viele `return`-Anweisungen enthalten. Anzustreben sind aber wenige `return`-Anweisungen, da sonst das Rücksprungverhalten unter Umständen schwer nachvollziehbar wird.
- Für den Rückgabewert findet gegebenenfalls eine implizite Typumwandlung statt.

Zum letzten Punkt soll folgendes Beispiel angeführt werden, welches zeigt, daß der Rückgabetyp einer Funktion stets geeignet zu wählen ist, um ungewollte Konvertierungsfehler bei Aufrufen zu vermeiden:

```
int quad2 ( double x, double y )
{
  return x*x + y*y;
}
  ...
double  erg, x, y;
x = y = 2.5;
erg = quad2( x, y ):
```

Der Wert von `x*x + y*y`, der in der Funktion `quad2` errechnet wird, beträgt $2,5^2 + 2,5^2 = 12,5$. Da der Rückgabetyp der Funktion `int` ist, wird vor der eigentlichen Rückgabe eine implizite Typumwandlung durchgeführt und der `int`-Wert 12 zurückgeben. Dieser wird dann der `double`-Variablen `erg` zugewiesen und dabei wird eine weitere implizite Typumwandlung durchgeführt, diesmal von `int` nach `double`.
Offenbar paßt der Rückgabetyp nicht zum Zweck der Funktion. Falls die eingetretene Wirkung tatsächlich erwünscht war, sollte die Funktion wie folgt definiert und aufgerufen werden:

```
double  quad2 ( double x, double y )
{
  return x*x + y*y;
}
  ...
double  erg, x, y;
x = 2.5;
y = 2.5;
erg = (int) quad2( x, y ):
```

In der letzten Zeile wird eine explizite Typumwandlung durchgeführt.

Ablauf eines Funktionsaufrufes

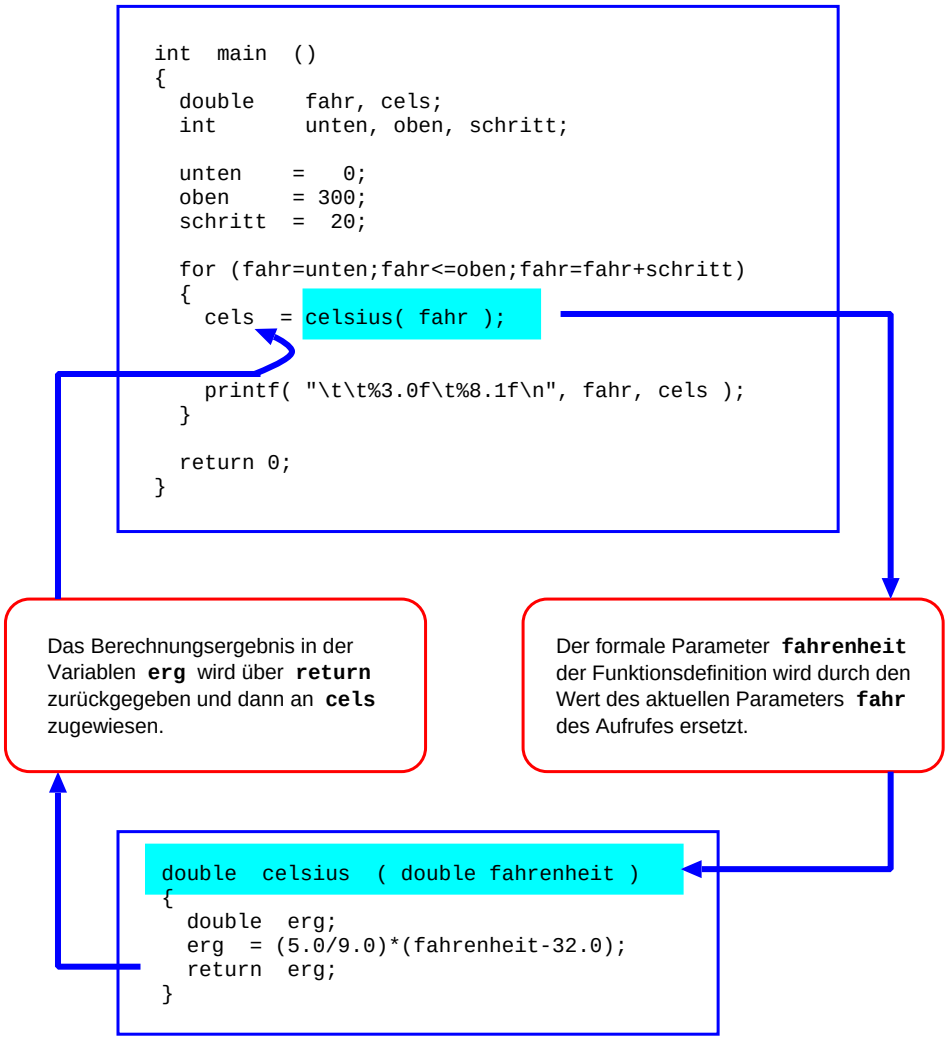


Abbildung 5.1.: Ablauf eines Funktionsaufrufes

In Abbildung 5.1 ist der grundsätzliche Ablauf eines Funktionsaufrufes gezeigt. Das Hauptprogramm (allgemein die aufrufende Umgebung) wird an der Stelle des Aufrufes verlassen und der Programmcode der Funktion `celsius` wird angesprungen. Vor dem Verlassen wird der Wert des an der Aufrufstelle übergebenen, aktuellen Parameters `fahr` ermittelt. Bei der Ausführung des Funktionscodes ersetzt der Wert des aktuellen Parameters den in der Funktionsdefinition verwendeten formalen Parameter `fahrenheit` (Platzhalter).

Der Code der Funktion wird mit einer `return`-Anweisung verlassen, die den ermittelten Rückgabewert mitnimmt. Der Rücksprung erfolgt in die aufrufende Umgebung hinter die Aufrufstelle. In der Abbildung 5.1 ist damit die rechte Seite der Zuweisung `cels = celsius( fahr )` bewertet, das heißt der Wert des Funktionsaufrufes kann nach links an die Variable `cels` zugewiesen werden.

5.2. Funktionsparameter mit Wertübergabe

Es gibt grundsätzlich zwei Arten, aktuelle Parameter an Funktionen zu übergeben:

- **Wertübergabe**. Die aktuellen Parameter werden ausgewertet und eine *Kopie des jeweiligen Wertes* wird an die aufrufende Funktion übergeben.
- **Adreßübergabe**. Die Speicheradressen der Parameter (Variablen, Felder, Strukturen, ...) werden an die aufrufende Funktion übergeben. Die Werte der Parameter werden aus der aufgerufenen Funktion heraus aus dem Originalspeicherplatz gelesen und können dort auch geändert werden.

In ***C*** ist die Wertübergabe der Standardfall, das heißt wenn nichts anderes angegeben wird, wird Wertübergabe durchgeführt. Die Adreßübergabe muß durch das Hinzufügen des Adreßoperators `&` zum Variablennamen erzwungen werden.

Da bei der Wertübergabe ein aktueller Parameter ausgewertet und dann eine Kopie dieses Wertes (in einer anonymen Speicherzelle) übergeben wird, ist es nicht möglich den Wert des aktuellen Parameters in der aufrufenden Umgebung zu ändern.
Diese wichtige Eigenschaft der Wertübergabe dient einerseits als Schutzeinrichtung, weil damit eine unbeabsichtigte Veränderung der aufrufenden Umgebung nicht möglich ist. Andererseits ist sie auch einschränkend, weil damit die Funktionsparameter nicht für die Rückgabe von Ergebnissen verwendet werden können. Diese Möglichkeit besteht nur, wenn Adreßübergabe verwendet wird.

Liste 5.1: Funktionsaufruf mit Wertübergabe

```
1  #include  <stdio.h>
2
3  int summe_1_bis_n ( int n )
4  {
5    int summe = 0;
6    do
7      summe += n;
8    while ( --n >0 );
9    return summe;
10 }       /* --------  end of function summe_1_bis_n  -------- */
11
12 int main ( int argc, char *argv[] )
13 {
14   int sum, n = 5;
15
16   sum = summe_1_bis_n ( n );                    /* n : Wertübergabe */
17
18   printf ("\nSumme 1 ... %d  =  %d\n", n, sum );
19
20   return 0;
21 }       /* --------  end of function main  -------- */
```

Die Unmöglichkeit, bei Wertübergabe eine Variable in der aufrufenden Umgebung zu ändern, wird in Liste 5.1 gezeigt. Die Variable `n` des Hauptprogrammes wird an die Funktion `summe_1_bis_n` übergeben. In dieser Funktion wird der formale Parameter `n` als Schleifenzähler verwendet und in der Bedingung der `do-while`-Schleife auf Null heruntergezählt. Das ist möglich, weil bei der Ausführung der Funktion der Wert der übergeben Größe `n` tatsächlich in einer anonymen Speicherzelle steht, die mit dem Speicherplatz der Variablen `n` im Hauptprogramm nichts zu tun hat.
Das Hauptprogramm gibt deshalb die Zeile

```
Summe 1 ... 5  =  15
```

aus und man erkennt, daß die Variable `n` des Hauptprogrammes ihren Wert 5 beibehalten hat.

5.3. Prototypen

Liste 5.2: Verwendung eines Prototyps

```
1  #include  <stdio.h>
2
3  double  celsius ( double fahrenheit );          /* Prototyp Funktion celsius */
4
5  /*---------------------------------------------------------------------------
6   *  Hauptprogramm
7   *--------------------------------------------------------------------------*/
8  int main  ( )
9  {
10   double    fahr, cels;                         /* Vereinbarungen            */
11   int       unten, oben, schritt;
12
13   unten   =   0;                                /* Tabellenanfang            */
14   oben    = 300;                                /* Tabellenende              */
15   schritt =  20;                                /* Schrittweite              */
16
17   /* ---------- Programmtitel, Tabellenkopf ---------- */
18   printf("\n\t*****  Umwandlung von Grad Fahrenheit in Grad Celsius  *****");
19   printf("\n\n\tGrad Fahrenheit\tGrad Celsius\n");
20
21   /* ---------- Tabellenzeilen ausgeben ------------------------------ */
22   for ( fahr=unten; fahr<=oben; fahr=fahr+schritt ) {
23     cels  = celsius( fahr );                    /* Aufruf Funktion celsius   */
24     printf( "\t\t%7.0lf\t%12.1lf\n", fahr, cels );
25     }
26   return 0;
27 }
28
29 /*---------------------------------------------------------------------------
30  *  Definition der Funktion celsius
31  *--------------------------------------------------------------------------*/
32 double  celsius ( double fahrenheit )
33 {
34   double  erg;                                  /* Hilfsvariable             */
35   erg = (5.0/9.0)*(fahrenheit-32.0);            /* Umrechnungsformel         */
36   return  erg;                                  /* Ergebnis zurückgeben      */
37 }
```

Funktionen müssen in ***C*** genau wie Variablen vor der ersten Verwendung vereinbart werden. Diese Forderung kann auf zwei Weisen erfüllt werden:

1. Alle Funktionsdefinitionen stehen vor den Aufrufen.
2. Für alle Funktionen werden vor dem ersten Aufruf Funktionsprototypen angegeben.

Die Auflistung aller Funktionsdefinitionen am Anfang einer Datei ist bei Programmen mit wenigen Funktionen einfach möglich und überschaubar. Bei umfangreichen Projekten ist diese Vorgehensweise nicht möglich oder mindestens zu aufwendig. Möglicherweise gibt es gar keine gültige Anordnung der Funktionsdefinitionen, weil sich die Funktionen gegenseitig aufrufen. In diesen Fällen müssen Funktionsprototypen verwendet werden.

> Der Prototyp einer Funktion besteht aus dem vollständigen Funktionskopf, der durch einen Strichpunkt abgeschlossen ist.

Prototypen erfüllen einen ähnlichen Zweck wie Variablenvereinbarungen. Sie ermöglichen dem Compiler die nachfolgenden Funktionsaufrufe auf die *formale Richtigkeit* hin zu überprüfen, weil die Prototypen bereits den Funktionsnamen, den Rückgabetyp, sowie die Anzahl und Datentypen der formalen Parameter enthalten.

Die eigentliche Funktionsdefinition kann dann an beliebiger Stelle in der Datei oder auch in einer anderen Datei stehen. Die Anordnung der Funktionsdefinitionen ist nun beliebig.
Es ist üblich, die Prototypen für alle aufzurufenden Funktionen an den Anfang der jeweiligen Datei zu schreiben. Liste 5.2 zeigt ein vollständiges Beispiel.

Selbstverständlich muß der Prototyp (Liste 5.2, Zeile 3) mit dem Funktionskopf (Zeile 32) genau übereinstimmen. Bei Änderungen sind dann natürlich beide Zeilen zu ändern.

Für Bibliotheksfunktionen gilt ebenfalls Vereinbarungszwang, das heißt für Bibliotheksfunktionen müssen ebenfalls Prototypen vor deren erster Verwendung angegeben werden. Diese Prototypen sind allerdings gruppenweise in eigenen Dateien, den sogenannten header-Dateien (Endung `.h`) zusammengefaßt und in Systemverzeichnissen abgelegt. Es genügt, eine `include`-Anweisung für die entsprechende `h`-Datei am Anfang des Programmes einzufügen, zum Beispiel Liste 5.2, Zeile 1. Der Präprozessor fügt vor der eigentlichen Übersetzung die gesamte `h`-Datei ein, die im Falle dieses Beispiels alle Prototypen für die Ein-/Ausgabefunktionen der Standardbibliothek enthält, zu denen auch der Prototyp für `printf` gehört.

Die Namen der zu den einzelnen Bibliotheksfunktionen gehörenden header-Dateien muß der jeweiligen Compiler- oder Systemdokumentation entnommen werden.

5.4. Gültigkeitsbereich und Lebensdauer von Bezeichnern

Bezeichner für Variablen, selbst definierte Datentypen und Funktionen können in einem Block, in einer Datei oder über Dateigrenzen hinweg sichtbar und damit zugreifbar sein. Die tatsächlich wirksamen Gültigkeitsbereiche müssen durch entsprechende Vereinbarungen oder durch die entsprechende Regeln beim Fehlen solcher Vereinbarungen festgelegt werden. Die Grundregel lautet:

> Bezeichner sind genau in dem Block gültig, in dem sie definiert sind.

Blöcke in diesem Sinn sind anonyme Blöcke (ein Paar geschweifter Klammern), Funktionsrümpfe und geklammerten Rümpfe von Schleifen und Verzweigungen (`if`, `switch`).

Geschachtelte und parallele Blöcke

Die folgenden Zeilen zeigen zwei geschachtelte Blöcke:

```
{                       /* ----- äußerer Block ----- */
  int a, b;
  ...

  {                       /* ----- innerer Block ----- */
    double  a, c;
    ...                   /* int a NICHT sichtbar */
```

```
                          /* int b ist sichtbar   */
  }

  ...                    /* double a, c NICHT mehr vorhanden */
                         /* int a ist wieder sichtbar        */
}
```

Zunächst gilt, daß die Variablen `a` und `b` im äußeren Block definiert und deshalb auch im gesamten äußeren Block und in dessen inneren Blöcken gültig sind.
Die Gültigkeit in inneren Blöcken ist solange gegeben bis ein Namenskonflikt auftritt. Das ist durch die Vereinbarung `double a, c` im inneren Block der Fall. Der Konflikt wird dadurch aufgelöst, daß die gleichnamige Variable des äußeren Blockes im gesamten inneren Block bis zu dessen Ende nicht mehr sichtbar ist.
Wird der innere Block verlassen, sind alle Namen des äußeren Blockes wieder sichtbar und alle Namen innerer Blöcke sind unbekannt, weil die Namen und Speicherplätze der dazugehörigen Variablen nicht mehr vorhanden sind.

Die Regeln sind entsprechend auf parallele Blöcke zu übertragen. Die Variablen in den folgenden beiden parallelen Blöcken gelten ausschließlich innerhalb derselben:

```
{             /* ----- 1. Block ----- */
  int a, b;
   ...
}

{             /* ----- 2. Block ----- */
  int a;      /* neues a  */
   ...        /* int a, b aus Block 1 sind NICHT mehr vorhanden */
}
```

Dieselben Verhältnisse gelten bei Funktionen. Funktionen sind gewissermaßen benannte Blöcke mit Parameter- und Rückgabeschnittstellen. Da es in ***C*** keine geschachtelten Funktionsdefinitionen gibt, stehen alle Funktionsdefinitionen auf der selben Stufe. Alle formalen Parameter und internen Bezeichner haben deshalb nur Gültigkeit innerhalb der jeweiligen Funktionsdefinition.

```
int reihensumme_1 ( double x, int n )
{
  int     i;
  double  summe = 0.0;
  ...
}

int reihensumme_2 ( double x, int n )
{
  int     i;
  double  summe = 0.0;
  ...
}

int main ( void )
{
  int     i;
  double  summe = 0.0;
  ...
}
```

Es ist deshalb ohne weiteres möglich und üblich, bei ähnlichen Funktionen gleichlautende formale Parameter und Variablen zu verwenden. Hier bestehen in keinem Fall Konflikte.

5.5. Speicherklassen

Jede Variable in *C* besitzt zwei Merkmale, den Datentyp und die Speicherklasse. Es gibt vier Speicherklassen und deshalb gibt es automatische, externe und statische Variablen, sowie Register-Variablen. Die Schlüsselwörter dazu lauten

```
auto      extern    static    register
```

5.5.1. Speicherklasse `auto`

Variablen die innerhalb von Funktionen oder Blöcken vereinbart werden, haben ohne zusätzliche Angaben die Speicherklasse `auto`. Das Schlüsselwort `auto` könnte bei der Vereinbarung verwendet werden, wird aber meist weggelassen.

```
auto int      a, b, c;
auto double   x, y, z;
```

Die wichtigsten Merkmale automatischer Variablen sind:

- Die Variablen sind nur innerhalb des Blockes gültig in dem sie vereinbart wurden.
- Der Speicherplatz wird für diese Variablen erst beim Betreten des Blockes beschafft. Beim Verlassen des Blockes wird der Platz wieder freigegeben.
- Der Inhalt des Speicherplatzes ist demnach nach dem Verlassen des Blockes nicht mehr zugreifbar.

5.5.2. Speicherklasse `extern`

Wenn eine Variable außerhalb einer Funktion vereinbart ist, dann heißt diese Variable extern. Ihr Speicherplatz ist dauernd vorhanden. Die Speicherklasse ist `extern`.

Externe Variablen sind für alle Funktionen sichtbar und zugreifbar, die selbst nach dieser Variablenvereinbarung definiert sind. Externe Variablen sind deshalb die einfachste und schnellste Möglichkeit, Werte zwischen Funktionen auszutauschen oder mehreren Funktionen gemeinsame Größen zur Verfügung zu stellen.

In dieser zunächst vorteilhaft erscheinenden Eigenschaft liegt jedoch die Gefahr, daß derartige Variablen unbeabsichtigten überschrieben werden können. Auch die Funktionen, die diese externen Variablen gar nicht benötigen, können auf deren Werte zugreifen und diese verändern. Dadurch entstehen besonders in großen Projekten schwer auffindbare Fehler. Genau aus diesem Grund muß der Gebrauch von externen Variablen auf den kleinstmöglichen Umfang eingeschränkt werden.

Liste 5.3: Gebrauch von externen (globalen) Variablen

```
1  #include  <stdio.h>
2
3  int a = 3, b = 2, c = 1;          /* globale Variablen */
4
5  int fkt (void);                   /* Prototyp          */
6
7  int   main (void)
8  {
9    printf("\na = %3d, b = %3d, c = %3d\n", a, b, c );
10
11   printf("\nRückgabewert von fkt() = %d\n", fkt() );
12
13   printf("\na = %3d, b = %3d, c = %3d\n", a, b, c );
14
15   return 0;
16 }       /* ----------  end of function main  ---------- */
17
18 int fkt (void)
19 {
20   int b, c;                       /* lokale Variablen                */
21
22   a = b = c = 8;                  /* globales a wird hier verändert! */
23   return (a+b+c);
24 }       /* ----------  end of function fkt  ---------- */
```

In Liste 5.3, Zeile 3 sind drei globale Variablen eingerichtet. Sie haben die Speicherklasse `extern` und sind in den beiden nachfolgend definierten Funktionen sichtbar. Die Ausgabe diese Programmes lautet

```
a =   3, b =   2, c =   1

Rückgabewert von fkt() = 24

a =   8, b =   2, c =   1
```

Die erste `printf`-Anweisung gibt die Variablenwerte so aus, wie sie in der Anfangswertzuweisung gesetzt wurden. In der zweiten `printf`-Anweisung wird die Funktion `fkt` aufgerufen. In dieser Funktion werden zwei gleichnamige `auto`-Variablen `b` und `c` vereinbart. Der Namenskonflikt wird dadurch aufgelöst, daß die gleichnamigen globalen Variablen innerhalb der Funktion `fkt` nicht sichtbar sind. Mit der globalen Variablen `a` besteht kein Konflikt. Sie erhält ebenfalls den Wert 8 und besitzt diesen natürlich auch noch im Hauptprogramm, wenn die Funktionsausführung bereits beendet ist.
Die letzte `printf`-Anweisung gibt den neuen Wert von `a` und die beiden unveränderten Werte von `b` und `c` aus.

Liste 5.4: Modul 1: Vereinbarung globaler Variablen

```
1  #include  <stdio.h>
2
3  int a = 3, b = 2, c = 1;          /* globale Variablen */
4
5  int fkt (void);                   /* Prototyp          */
6
7  int   main (void)
8  {
9    printf("\na = %3d, b = %3d, c = %3d\n", a, b, c );
10
11   printf("\nRückgabewert von fkt() = %d\n", fkt() );
12
13   printf("\na = %3d, b = %3d, c = %3d\n", a, b, c );
14
15   return 0;
16 }       /* ----------  end of function main  ---------- */
```

Liste 5.5: Modul 2: Verwendung externer Variablen (aus Modul 1)

```
1  extern int a;                     /* globale Variable in einer anderen Datei*/
2
3  int fkt (void)
4  {
5    int b, c;                       /* lokale Variablen                       */
6
7    a = b = c = 8;                  /* globales a wird hier verändert!        */
8    return (a+b+c);
9  }       /* ----------  end of function fkt  ---------- */
```

Die Listen 5.4 und 5.5 zeigen eine Zerlegung des Programmes in Liste 5.3 in zwei getrennt übersetzbare Teile. Im zweiten Teil (Liste 5.5, Zeile 1) muß die Variable `a` mit dem Schlüsselwort `extern` versehen werden. Der Compiler erkennt daran, daß diese Variable in einer anderen Datei vereinbart ist, und begnügt sich bei der Übersetzung mit der Überprüfung der typgerechten Verwendung.
Die Möglichkeit, getrennt übersetzbare Programmteile mit derartigen Querbezügen zu verwenden, ist eine wichtige Voraussetzung für die Erstellung großer Programme.

5.5.3. Speicherklasse `static`

Lokale `static`-Variablen

Die Zusatzvereinbarung `static` für eine lokale Variable bewirkt, daß der Speicherplatz dieser Variablen beim Verlassen des Blockes erhalten bleibt. Die Funktion `next_fibonacci` ist ein Generator für die Fibonacci-Zahlen $f_1, f_2, f_3, \ldots$. Die Fibonacci-Folge ist wie folgt definiert:

$$f_n = f_{n-1} + f_{n-2} \quad für \quad n \geq 2\,; \qquad Anfangswerte: f_0 = 0, \quad f_1 = 1 \tag{5.1}$$

Diese Folge beginnt demnach mit den Werten $0, 1, 1, 2, 3, 5, 8, 13, \ldots$, das heißt der nächste Wert errechnet sich als Summe der beiden vorhergehenden Werte.

```
unsigned long next_fibonacci ( )
{
  static  unsigned long fnm2 = 0;
  static  unsigned long fnm1 = 1;
          unsigned long erg  = fnm1;
  fnm1  = erg + fnm2;
  fnm2  = erg;
  return erg;
}   /* -----  end of function next_fibonacci  ----- */
```

Der erste Aufruf liefert die Fibonacci-Zahl f_1 (= 1), der zweite Aufruf f_2 (= 2) und so weiter. Die Initialisierungen mit 0 und 1 werden nur im ersten Aufruf verwendet. Die letzten beiden Fibonacci-Zahlen werden in den `static`-Variablen `fnm2` und `fnm1` zur Verwendung im nächsten Aufruf gespeichert.

In der nachfolgenden Funktion wird ein Feld dazu verwendet, die Monatslänge in Tagen zur weiteren Verwendung in der Funktion zu speichern. Da das Feld statisch ist, muß es nicht jedesmal initialisiert werden, wenn die Funktion aufgerufen wird. Das erste Element ist lediglich ein Platzhalter, damit auf die Monatslängen mit den üblichen Monatsnummern (Januar = 1 usw.) zugegriffen werden kann.

```
void do_something ( )
{
  static unsigned int monatslaenge[]  = { 0,              /* Platzhalter */
                                          31, 28, 31, 30, /* Jan ... Apr */
                                          31, 30, 31, 31, /* Mai ... Aug */
                                          30, 31, 30, 30, /* Sep ... Dez */
                                          };
  /* ... */
}   /* -----  end of function do_something  ----- */
```

Globale `static`-Variablen

Werden globale Variablen mit der Zusatzvereinbarung `static` versehen, dann ist ihre Gültigkeit trotz ihres globalen Charakters auf die Datei beschränkt, in der diese Variablen vereinbart sind.

Liste 5.6: Modul 3: Vereinbarung einer globalen `static`-Variablen

```
1  #include  <stdio.h>
2
3  static  int x=7;                    /* globale Variable; NICHT exp.  */
4          int a=3, b=2, c=1;          /* globale Variablen; exportiert */
5
6  int fkt (void);                     /* Prototyp                      */
7
8  int   main (void)
9  {
10   printf("\na = %3d, b = %3d, c = %3d\n", a, b, c );
11   printf("\nRückgabewert von fkt() = %d\n", fkt() );
12   printf("\na = %3d, b = %3d, c = %3d\n", a, b, c );
13   return 0;
14 }       /* ----------  end of function main  ---------- */
```

Liste 5.7: Modul 4: Verwendung einer externen Variablen (aus Modul 3)

```
1  extern int a;                        /* globale Variable in einer anderen Datei*/
2  extern int x;                        /* globale Variable in einer anderen Datei*/
3
4  int fkt (void)
5  {
6    int b, c;                          /* lokale Variablen                       */
7    c = x;                             /* erzeugt Fehler beim Binden !           */
8    a = b = 8;                         /* globales a wird hier verändert !       */
9    return (a+b+c);
10 }       /* ----------  end of function fkt  ---------- */
```

In Liste 5.6, Zeile 3, wird eine globale `static`-Variable `x` eingerichtet, deren Gültigkeitsbereich auf diese Datei beschränkt ist. Die Datei in Liste 5.7 läßt sich fehlerfrei einzeln übersetzen, weil der Compiler beim Übersetzen nicht erkennen kann, in welcher anderen Datei und mit welcher Zusatzvereinbarung diese Variable vereinbart ist. Der Binder stellt allerdings dann fest, daß die Variable `x` von keiner anderen zum Gesamtprogramm gehörenden Datei exportiert wird. Das Programm kann nicht gebunden werden!

5.5.4. Speicherklasse `register`

Die Zusatzvereinbarung `register` fordert den Compiler auf, die zugehörige Variable in einem CPU-Register abzulegen. Das geschieht in der Hoffnung, daß das Programm dadurch schneller ausgeführt wird. Der Compiler muß sich jedoch nicht an diese Aufforderung halten.

```
int summe_1_bis_n ( int n )
{
  register int  i, summe = 0;
  for( i=1; i<=n; i++ )
    summe += n;
  return summe;
}       /* --------  end of function summe_1_bis_n  -------- */
```

5.5.5. Speicherklasse `volatile`

Die Zusatzvereinbarung `volatile` gibt an, daß die zugehörige Variable nicht in einem CPU-Register abgelegt werden darf. Das geschieht zum Beispiel, weil die Variable von mehreren parallel laufenden Programmsträngen (engl. *threads*) verwendet wird oder von der Hardware asynchron verändert werden kann. Der Compiler unterdrückt außerdem alle Optimierungen für diese Variablen (zum Beispiel Umordnung von Rechenausdrücken, kurzzeitige Speicherung in einem Register). Im folgenden Beispiel wird in jedem Schleifendurchlauf der Speicherplatz der Variablen `flag` gelesen und nicht etwa eine Kopie des Wertes in einem Register. Diese würde sich nicht ändern und die Schleife würde deshalb nicht beendet werden.

```
volatile int flag = 1;

void DoSomething ( )
{
  while( flag == 1 )
    sleep(1);

  /* hier weiter */
}       /* --------  end of function DoSomething  -------- */
```

5.6. Rekursion

Eine Funktion heißt *rekursiv*, wenn sie sich direkt oder indirekt selbst aufruft. Bei einem indirekten Aufruf ruft die rekursive Funktion zunächst mindestens eine andere Funktion auf, die dann ihrerseits die Ausgangsfunktion aufruft.

direkte Rekursion	Funktion `f` ruft Funktion `f` auf
indirekte Rekursion	Funktion `f` ruft Funktion `g` auf Funktion `g` ruft Funktion `f` auf

Liste 5.8: Rekursive Berechnung der Fakultät einer ganzen Zahl

```
1 unsigned int fakultaet ( unsigned int n )
2 {
3   if ( n <= 1 )                                  /* Ende: 0! = 1! = 1 */
4     return 1;
5   return n*fakultaet(n-1);                       /* rekursiver Aufruf */
6 }       /* ----------  end of function fakultaet  ---------- */
```

Liste 5.8 zeigt die rekursive Funktion `fakultaet`, die die Fakultät einer ganzen Zahl n berechnet: $n! = n \cdot (n-1) \cdot \ldots \cdot 2 \cdot 1$. Definitionsgemäß ist $0! = 1$. Für negative Zahlen ist die Fakultät nicht definiert. Aus der Definition der Rekursion folgt

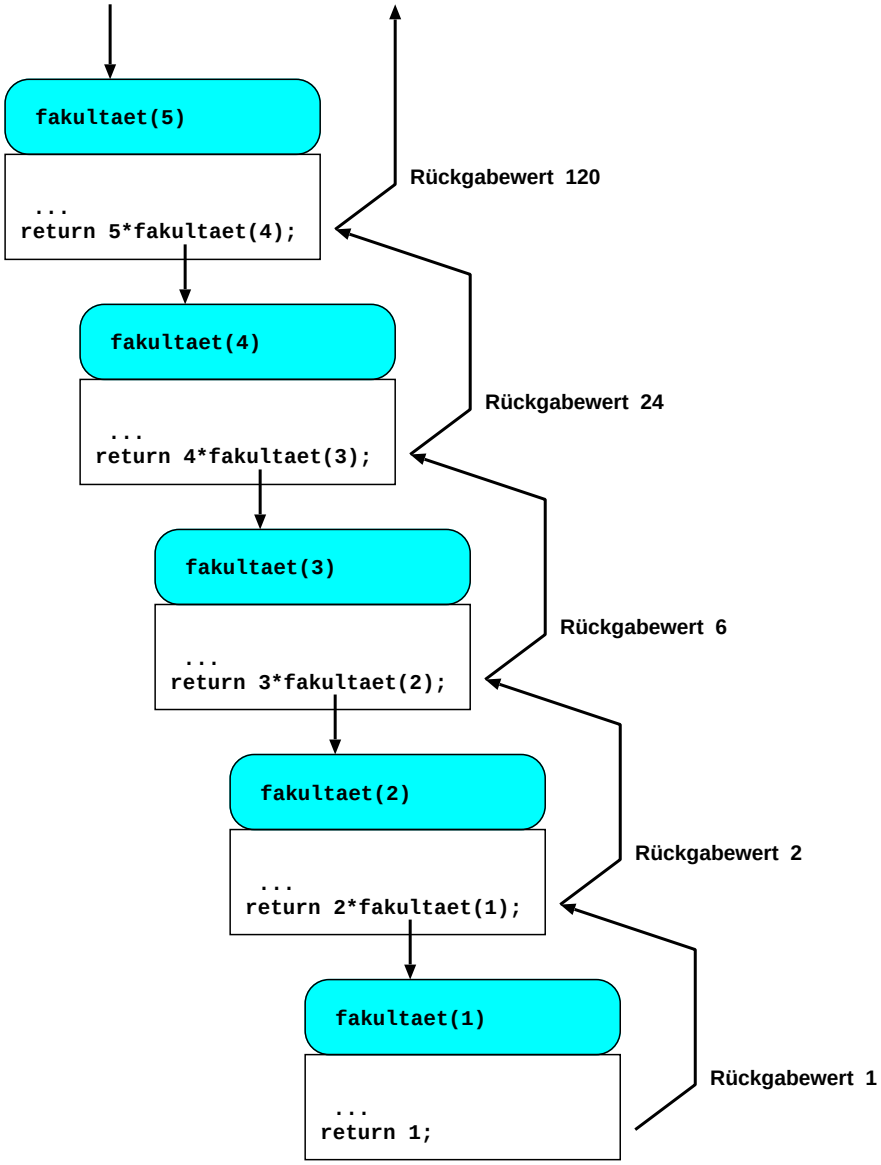
$$\begin{aligned} n! &= n \cdot (n-1) \cdot \ldots \cdot 2 \cdot 1 \\ n! &= n \cdot [(n-1) \cdot \ldots \cdot 2 \cdot 1] \\ n! &= n \cdot (n-1)! \end{aligned}$$

In der vorletzten Zeile ist die Fakultät durch sich selbst definiert. Das kann nur sinnvoll sein, wenn die Rekursion durch eine nichtrekursive Zusatzfestlegung beendet werden kann. In diesem Fall ist das $0! = 1! = 1$.
Die Funktion `fakultaet` besitzt einen für rekursive Funktionen typischen Aufbau. Vor der eigentlichen Rekursion steht in einer rekursiven Funktion immer die Überprüfung, ob die Rekursion beendet werden muß (Zeile 3). Wenn der Parameter `n` den Wert 0 oder 1 besitzt wird die Funktion mit dem Wert für $0! = 1! = 1$ beendet (Zeile 4). Wenn das nicht der Fall ist erfolgt eine weitere Anwendung des Rekursionsschemas. Die Funktionsausführung gelangt zu der Anweisung (Zeile 5)

```
return n*fakultaet(n-1);
```

Um den Rückgabewert zu berechnen ruft die Funktion `fakultaet` sich selbst auf, allerdings mit dem um 1 verminderten Wert des Eingabeparameters `n`. Da die `return`-Anweisung einen Funktionsaufruf enthält, kann die Funktion erst dann verlassen werden, wenn der Rückgabewert dieses Aufrufes als Zahlenwert berechnet wurde.
Wenn der Wert von `n-1` in dem in der `return`-Anweisung erzeugten Aufruf größer 1 ist, dann wird ein weiterer rekursiver Aufruf mit einem erneut um 1 verminderten Aufrufparameter stattfinden. Der Vorgang wird solange fortgesetzt, bis der Aufrufparameter den Wert 1 besitzt. Dann erfolgt die Beendigung der Funktion über die Abbruchbedingung (Zeilen 3,4).

Abbildung 5.2 zeigt den tatsächlichen Ablauf. Zur Berechnung von 5! werden durch den Aufruf von `fakultaet(5)` vier weitere Funktionsaufrufe erzeugt. Die ersten vier können zunächst nicht beendet werden, da jeweils zunächst ein Rückgabewert in der eigenen `return`-Anweisungen empfangen werden muß. Erst der letzte Aufruf (`fakultaet(1)`) beendet die



Die Variablen in den `return`-Anweisungen wurden zur Verdeutlichung durch ihre jeweiligen Werte ersetzt.

Abbildung 5.2.: Rekursive Funktionsaufrufe bei der Berechnung von 5!

Rekursion und gibt den Wert 1 an den vorletzten Aufruf zurück. Nun kann dieser Aufruf mit dem Rückgabewert $2 \cdot 1$ beendet werden und so weiter. Am Ende kann der erste Aufruf (`fakultaet(5)`) beendet werden. Dieser gibt das Ergebnis $5! = 120$ zurück.

Da jeder Funktionsaufruf Speicher und Rechenzeit verbraucht, wird die Rekursion hauptsächlich dann verwendet, wenn andere Lösungen wesentlich aufwendiger zu programmieren sind. Die rekursive Berechnung einer Fakultät ist zwar gut geeignet um das Prinzip der rekursiven Programmierung zu erläutern, ist aber ansonsten keine typische Anwendung einer Rekursion. Die Fakultät läßt sich natürlich wesentlich einfacher und schneller mit Hilfe einer Schleife berechnen. Typische Anwendungen der Rekursion sind das Durchlaufen komplexer Datenstrukturen (zum Beispiel Bäume oder Graphen), die Erzeugung von Permutationen oder die Lösung von Optimierungsproblemen (zum Beispiel Stundenplanerstellung, optimale Maschinenbelegung und ähnliches).

6. Felder und Zeiger

6.1. Eindimensionale Felder

Wie bereits in Abschnitt 1.9 dargestellt wurde, dienen Felder zum Abspeichern von mehreren Variablen mit *demselben Datentyp und Namen*. Die Unterscheidung der einzelnen Feldelemente geschieht durch Indices. Der Index des ersten Feldelementes ist immer 0. Zur Adressierung der einzelnen Elemente werden ganzzahlige Indexausdrücke verwendet.

Felder treten meistens in Verbindung mit Schleifen auf, mit deren Hilfe der Durchlauf durch ein gesamtes Feld oder durch einen Teilbereich bewerkstelligt wird. In Liste 1.11 wurde die Verwendung eines Feldes in einem vollständigen Programm bereits gezeigt.

Gelegentlich ist es sinnvoll, einem Feld eine feste Anfangswertbelegung bei der Vereinbarung zuzuweisen. Das geschieht in der folgenden Weise durch die Zuweisung einer geklammerten Liste von Anfangswerten:

```
int a[4] = { 1, 2, 4, 9 };
int b[4] = { 3 };
int c[ ] = { 2, 3, 5, 7, 11, 13, 17 };
```

Feld `a` wird beginnend mit `a[0] = 1` mit den vier angegebenen Werten initialisiert. In Feld `b` erhält nur das erste Element `b[0]` den Anfangswert 3, die restlichen Elemente werden mit Null initialisiert, da weitere Angaben fehlen. Feld `c` ist ohne ausdrückliche Längenangabe vereinbart. Die Feldlänge wird auf Grund der Länge der Initialisierungsliste auf 7 festgesetzt und die angegeben Werte entsprechend zugewiesen. Die Liste kann erweitert werden, ohne eine redundante Längenangabe ebenfalls ändern zu müssen.

Manchmal ist es erforderlich, die Länge eines Feldes wie `c` festzustellen, zum Beispiel um diese Länge zur Schleifensteuerung verwenden zu können. Das geschieht mit Hilfe des Operators `sizeof()`:

```
int c[ ] = { 2, 3, 5, 7, 11, 13, 17 };
int nc   = sizeof( c ) / sizeof( int );        /* Länge des Feldes c */
```

`sizeof` liefert die Länge eines Speicherobjektes oder eines Datentyps in Byte (siehe auch Abschnitt 6.7). Werden 4 Bytes zur internen Speicherung einer `int`-Größe verwendet, dann liefert die Berechnung vom `nc` als Feldgröße $28/4 = 7$ Elemente.

6.2. Mehrdimensionale Felder

Die Vereinbarung mehrdimensionalen Felder erfolgt durch die Angabe weiterer Längen:

```
int a[10];                    /*  1 Zeile                          */
int b[10][20];                /* 10 Zeilen, 20 Spalten             */
int c[10][20][30];            /* 10 Zeilen, 20 Spalten, 30 Elem. Tiefe */
```

Das Feld `b` ist zum Beispiel zur Abspeicherung einer 10×20-Matrix geeignet. Das Feld `c` könnte eine Temperaturverteilung in einem Quader von $10 \times 20 \times 30 = 6000$ Elementen aufnehmen.

Die Bearbeitung mehrdimensionaler Felder geschieht in der Regel mit geschachtelten Schleifen. Der folgende Abschnitt zeigt die Belegung eines Feldes mit einer Einheitsmatrix:

```
int     i, j;                          /* Schleifenzähler                */
int     n = 8;
double  mat[n][n];                     /* n*n-Matrix                     */

for ( i=0; i<n; i+=1 )                 /* Schleife über die Zeilen       */
  for ( j=0; j<n; j+=1 )               /* Schleife über die Spalten      */
    mat[i][j] = 0.0;

for ( i=0; i<n; i+=1 )                 /* Schleife über die Diagonalelem. */
    mat[i][i] = 1.0;
```

Das erste Paar geschachtelter Schleifen setzt alle Matrixwerte zunächst zu Null. Die anschließende Schleife setzt dann nur die Diagonalelemente zu Eins.

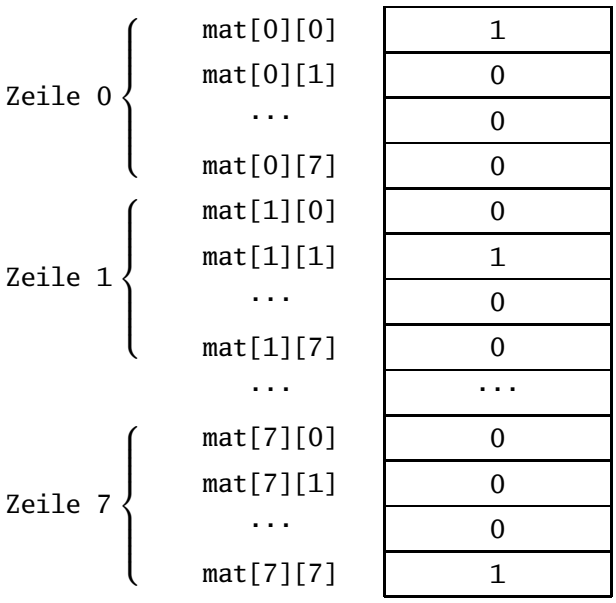


Abbildung 6.1.: Linearisierte Darstellung einer 8×8-Einheitsmatrix im eindimensionalen Speicher

Die Abbildung 6.1 zeigt die Speicherdarstellung der 8×8-Einheitsmatrix aus dem obenstehenden Beispiel im eindimensionalen Speicher: die Zeilen der Matrix sind hintereinander angeordnet, die Elemente stehen fortlaufend in aufeinanderfolgenden Speicherplätzen.

Auch mehrdimensionale Felder können bei der Vereinbarung mit Anfangswerten versehen werden. Die folgende 3×3-Matrix `rot11` wird durch eine Liste bestehend aus drei Zeilen initialisiert:

```
double  rot11[3][3] = { { +1.0,  0.0,  0.0 },          /* Zeile 0 */
                        {  0.0,  0.0, -1.0 },          /* Zeile 1 */
                        {  0.0, +1.0,  0.0 } };        /* Zeile 2 */
```

6.3. Zeiger

Ein Zeiger ist eine Variable, die eine Adresse als Wert enthält.

Zeiger dienen unterschiedlichen Zwecken. Zum einen ist es oft bequem oder erforderlich, auf Datenstrukturen mittels Zeiger zuzugreifen. Das geschieht unter anderem bei der Übergabe umfangreicher Datenstrukturen an Funktionen (Abschnitt 6.4), um den Aufwand des Kopierens bei einer Wertübergabe zu vermeiden. Wenn der Speicherbedarf zur Ablage von Daten erst zur Laufzeit eines Programmes ermittelt werden kann, weil das Programm zum Beispiel Bilder völlig unterschiedlicher Größe verarbeiten soll, dann kann der Speicher dynamisch, das heißt zur Laufzeit des Programmes, vom Betriebssystem angefordert werden. Das Programm erhält in diesem Falle die Anfangsadresse eines ansonsten namenlosen Speicherbereiches zurück, die in einer Adreßvariablen, einem Zeiger, abgelegt wird (Abschnitt 6.7). Viele Bibliotheksfunktionen nutzen aus diesen Gründen ebenfalls Zeiger. Zunächst soll ein einfaches Beispiel betrachtet werden:

```
int  a;                 /* int-Variable                */
int *pa;                /* Zeiger auf eine int-Größe   */

a   = 77;               /* Wert 77 (= 1001101) zuweisen */
pa  = &a;               /* Adresse von a an pa zuweisen */
```

Die `int`-Variable `a` wird vereinbart und erhält den Wert 77. Zusätzlich wird die Variable `pa` vereinbart. `pa` wird durch den Zusatz `*` als Zeiger auf eine `int`-Größe gekennzeichnet, der Datentyp ist also `int*` . Zeiger sind in ***C*** typbehaftet, der Stern steht üblicherweise beim Variablennamen.
Die Zeigervariable `pa` besitzt nach der Vereinbarung keinen Wert. Durch die Zuweisung `pa = &a` erhält `pa` die Speicheradresse der Variablen `a` als Wert zugewiesen. Der Adreßoperator `&` beschafft hierbei die Adresse der Variablen `a`. Die nachfolgende Abbildung zeigt die Verhältnisse im Speicher (die Adressen sind natürlich Beispielwerte).

Name	Typ	Adresse	Inhalt
			...
a	int	bffff284	1001101
			...
pa	int*	bffff280	bffff284
			...

Abbildung 6.2.: Variable `a` und Zeiger `pa` auf diese Variable im Speicher

Man erkennt, daß der Speicherinhalt der Variablen `pa` die Adresse der Variablen `a` ist. Die Variable `pa` selbst liegt natürlich unter einer anderen Adresse im Speicher (hier: `bffff280`).

Zeiger dienen in erster Linie dazu, auf den Inhalt der in ihnen gespeicherten Adressen zuzugreifen. Dazu muß der Inhaltsoperator `*` verwendet werden. Mit seiner Hilfe kann auf den Inhalt (Wert) des Speicherplatzes zugegriffen werden, auf den der Zeiger zeigt. Diesen Vorgang

nennt man auch Dereferenzierung des Zeigers.
Nachdem im obenstehenden Beispiel der Zeiger `pa` die Adresse von `a` erhalten hat, liefert der Ausdruck `*pa` den Inhalt der Adresse in `pa`, also den Wert 77. Das folgende Programm (Liste 6.1) gibt Werte und Adressen zu diesem Beispiel aus.

Liste 6.1: Direkte und indirekte Ausgabe von Adressen und Werten

```
1  #include  <stdio.h>
2
3  int main (void)
4  {
5    int  a  = 77;                                  /* int-Variable a       */
6    int *pa = &a;                                  /* Zeiger auf a         */
7
8    printf("\n  a = %d",   a );                    /* Wert der Variablen a */
9    printf("\n*pa = %d", *pa );                    /* Wert des Variablen a */
10   printf("\n pa = %p",  pa );                    /* Wert des Zeigers     */
11   printf("\n &a = %p",  &a );                    /* Adr. der Variablen a */
12   printf("\n&pa = %p", &pa );                    /* Adr. des Zeigers pa  */
13   printf("\n\n");
14
15   return 0;
16 }       /* ----------  end of function main  ---------- */
```

Die Programmausgabe liefert die Zeilen

```
  a = 77
*pa = 77
 pa = bffff284
 &a = bffff284
&pa = bffff280
```

und zeigt die Verwendung des Adreßoperators und des Inhaltsoperators. Das Teilformat `%p` gibt einen Zeiger als hexadezimale Adresse aus. Die mit Hilfe dieser Operatoren gebildeten Ausdrücke sind nachfolgend zusammengestellt:

Ausdruck	**Bedeutung**
`a`	Wert der Variable `a`
`&a`	Adresse der Variablen `a`
`pa`	Zeiger auf `a` (= Adresse der Variable `a`)
`*pa`	Inhalt der Adresse der Variable `a` (= Wert von `a`)
`&pa`	Adresse der Zeigervariablen `pa`

6.4. Funktionsparameter mit Adreßübergabe

In Abschnitt 5.2 wurden Funktionsparameter mit Wertübergabe betrachtet. Diese Art der Übergabe ist sicher, da eine Funktion die so übergebenen Parameter nicht für die aufrufende Umgebung ändern kann. Andererseits ist die Wertübergabe genau aus diesem Grund einschränkend.

Die zweite Möglichkeit einen Parameter an eine Funktion zu übergeben ist die Adreßübergabe. Hier wird nicht eine Kopie des Parameterwertes übergeben, sondern die *Speicheradresse des Parameters* . Dadurch besteht aus dem Innern einer Funktion heraus die Möglichkeit, auf den

Speicherplatz des Parameters zuzugreifen und damit auch den Wert zu ändern. Nach Verlassen der Funktion bleibt damit diese Änderung erhalten. Mit der Adreßübergabe ist es nun möglich, einen Wert in einer Funktion zu ändern oder auch erst festzulegen und das Ergebnis in die aufrufende Umgebung zu exportieren.
Damit stehen nicht nur sogenannte Eingabeparameter für eine Funktion zur Verfügung, sondern auch *Ausgabeparameter*. Dadurch werden die Verwendungsmöglichkeiten von Funktionen wesentlich erweitert.
Das soll am Beispiel des Wertetausches verdeutlicht werden. Gelegentlich ist es erforderlich, daß zwei Variablen oder Feldplätze ihre Werte tauschen. Die übliche Lösung verwendet einen Ringtausch mit einer Hilfsvariablen:

```
int a, b, h;
 ...
h = a;        /* Wertetausch a <=> b */
a = b;
b = h;
```

Wenn der Wertetausch öfters vorkommt, ist die Einrichtung einer Funktion `tausche` sinnvoll. Da eine derartige Funktion zwei Eingabeparameter (hier `a` und `b`) benötigt und auch beide Größen verändert, ist eine Lösung mit den bisherigen Mitteln nicht möglich. Hier die Lösung, die eine Funktion mit Adreßübergabe verwendet:

```
void  tausche ( int *a, int *b ) {
  int h;
   h  = *a;
  *a  = *b;
  *b  =  h;
}
```

In der Parameterliste sind die Parameter `a` und `b` als *Zeiger* vereinbart. Der Zugriff auf den Inhalt der übergebenen Adressen geschieht, wie oben bereits gezeigt, mit Hilfe des Inhaltsoperators. Nach dem Ringtausch sind die Werte ausgetauscht. Beim Aufruf der Funktion müssen jetzt natürlich auch Adressen übergeben werden:

```
int a, b;
 ...
tausche ( &a, &b );     /* Adreßübergabe !                  */
 ...                    /* Werte von a und b sind getauscht */
```

Wegen des Adreßzugriffes sind die Werte der beiden Variablen in der aufrufenden Umgebung nach dem Funktionsaufruf getauscht.

Zur Handhabung der Adreßübergabe bei Funktionen sind also die Parameter als Zeiger zu vereinbaren und im Innern der Funktion mit Hilfe des Inhaltsoperators auf die Werte zuzugreifen. Beim Aufruf müssen Adressen übergeben werden.

Inhaltsoperator `*`	*In einer Vereinbarung* zeichnet der Inhaltsoperator eine Größe als Zeiger aus. *Außerhalb einer Vereinbarung* erzwingt der Inhaltsoperator einen Adreßzugriff.
Adreßoperator `&`	Der Adreßoperator liefert stets die Speicheradresse einer Größe.

Adreß- und Wertübergabe können gemischt auftreten, wie das nachfolgende Beispiel der Funktion `polar_kartesisch` zeigt, die Polarkoordinaten in kartesische Koordinaten umrechnet.

Für die Umrechnung gilt

$$\begin{aligned} x &= r \cdot \cos(\phi) \\ y &= r \cdot \sin(\phi) \end{aligned}$$

Der Radius r und der Winkel ϕ sind Eingabeparameter, die Koordinatenwerte x und y sind Ausgabeparameter. Die Lösung sieht wie folgt aus:

```
void  polar_kartesisch ( double  r, double  phi,      /* Eingabeparameter */
                         double *x, double *y   )     /* Ausgabeparameter */
{
  *x  = r*cos(phi);
  *y  = r*sin(phi);
}
```

6.5. Felder und Zeiger

Feldnamen sind in *C* gleichzeitig Zeiger mit konstanten Adressen:

Ein Feldname kann als Zeiger auf das erste Feldelement verwendet werden.

Damit kann zum Beispiel zur Feldbearbeitung einer Zeigervariablen die Adresse des ersten Feldelementes zugewiesen werden. Voraussetzung einer sinnvollen Verwendung ist die Möglichkeit zur Adreßrechnung. So können Zeiger mit Hilfe der Inkrement- und Dekrementoperatoren um einen Adreßwert erhöht und vermindert werden. Weiterhin können Zeigerwerte durch Addition und Subtraktion ganzzahliger Größen verändert werden.

```
double  a[10];
double  *pa1, *pa2;

pa1 = a;            /* Adresse von a[0] zuweisen */
pa2 = pa1 + 3;      /* Adresse von a[3] bilden   */
```

In den obenstehenden Zeilen wird der Feldname `a` als Adresse verwendet und diese dem Zeiger `pa1` zugewiesen. Der Zeiger `pa2` enthält nach der Addition der Konstanten 3 die Adresse des Feldelementes `a[3]`. Da Zeiger einen Typ haben (hier „Zeiger auf `double`“) zeigt die Adresse in `pa2` im Speicher $3 \cdot 8$ Byte weiter als die Adresse in `pa1`, weil ein einzelnes `double`-Feldelement 8 Byte Speicherplatz benötigt. Bei einem `char`-Feld würden die Adressen entsprechen dem Speicherbedarf von 1 Byte pro Feldelement um 3 Byte hochgesetzt.
Der Programmierer kann bei der Adreßrechnung also in logischen Adressen denken und muß sich um die typgerechte Adreßbildung nicht kümmern.

Liste 6.2 zeigt die Verwendung von Zeigern zur Feldadressierung. In Zeile 9 wird der Zeiger `pa` auf das Feldelement `a[0]` gesetzt. Der Ausdruck `*pa++` in Zeile 11 ist der Inhalt des Feldelementes auf den der Zeiger `pa` gerade zeigt. Der Zeiger wird nach der Zuweisung inkrementiert. In Zeile 13 wird der Zeiger `pa` als Schleifenzähler verwendet. Die Initialisierung setzt den Zeiger zunächst auf das Feldelement `a[0]`. Das Abbruchkriterium (`pa < a+n`) verlangt, daß die Adresse kleiner bleibt als die Adresse des (nicht vorhandenen!) Feldelementes `a[n]`. Der Zeiger wird nach jedem Schleifendurchlauf um eine Adresse erhöht.

Zeiger- und Indexzugriff können bei Feldern gleichwertig eingesetzt werden. Die beiden Summationen

```
summe = summe + a[i];
pa    = a;
summe = summe + *(pa+i);
```

sind gleichwertig. Die runden Klammern in der letzten Zeile sind zwingend: zunächst wird die Adresse `pa+i` gebildet, dann erst wird auf den Inhalt dieser Adresse zugegriffen.

Liste 6.2: Verwendung von Zeigern zur Feldadressierung

```
1  #include  <stdio.h>
2
3  int main (void)
4  {
5    int  i, n = 10, summe = 0;
6    int  a[n];                                  /* Feld mit n Elementen       */
7    int *pa;
8
9    pa  = a;                                    /* Zeiger initialisieren      */
10   for( i=0; i<n; i++ )
11     *pa++ = i*i;                              /* Feldelemente besetzen      */
12
13   for( pa=a; pa < a+n; pa++ )                 /* Zeiger als Schleifenzähler */
14     summe += *pa;                             /* Feldelemente aufsummieren  */
15
16   printf("\n\tSumme = %d\n\n", summe );
17   return 0;
18 }       /* ----------  end of function main  ---------- */
```

6.6. Felder als Funktionsargumente

Eindimensionale Felder

Felder werden grundsätzlich mittels Adreßübergabe an Funktionen übergeben. Bei einer Wertübergabe wäre eine Kopie des jeweiligen Feldes zu erzeugen; das verbietet sich wegen des Aufwandes, insbesondere bei umfangreichen Feldern. In einer Funktionsdefinition muß der Feldparameter deshalb als Zeiger angegeben werden. Dazu kann die Zeigerschreibweise verwendet werden, oder, bei eindimensionalen Feldern, ein Paar leerer eckiger Klammern, die dem Parameternamen nachgestellt sind:

```
double summe ( double *a  , int n ) { ... }
double summe ( double  a[], int n ) { ... }
```

Diese beiden Schreibweisen sind gleichwertig. Beim Aufruf können beliebige, sinnvolle Adressen übergeben werden. Im einfachsten Fall ist das der Name eines Feldes, zum Beispiel `fld`. Festlegung der Adresse durch Adreßrechnung ist ebenfalls möglich:

```
int     n = 1000;
double  fld[n];
  ...
erg = summe(  fld    ,  n );                  /* fld[  0] + ... + fld[999] */
erg = summe( &fld[10], 11 );                  /* fld[ 10] + ... + fld[ 20] */
erg = summe(  fld + 5,  3 );                  /* fld[  5] + ... + fld[  7] */
```

Unter der Voraussetzung, daß die Funktion summe die in ihrem zweiten Parameter angegebene Anzahl von Feldelementen aufsummiert, werden durch die letzten drei Zeilen die in den Kommentaren angegebenen Summen gebildet.

Mehrdimensionale Felder

Die Übergabe mehrdimensionaler Felder an Funktionen ist ebenfalls möglich. Hierbei sind zwei Fälle zu unterschieden. Sollen Felder übergeben werden, deren Größe bereits zur Übersetzungszeit feststeht, dann müssen die Felddimensionen in der Parameterangabe der Funktionsdefinition erscheinen. Die nachfolgende Liste zeigt das am Beispiel der Funktion summe2. Der erste Parameter hat den Namen a und ist stets als zweidimensionales Feld der Größe N1×N2 zu übergeben. Beim Aufruf genügt dann der Name des tatsächlich zu übergebenden Feldes.

```
#define N1  200                                 /* max. Zeilenanzahl  */
#define N2  100                                 /* max. Spaltenanzahl */

double summe2 ( double a[N1][N2], int n1, int n2 )
{
  ...
}       /* ----------  end of function summe2  ---------- */

int main ( void )
{
  int n1, n2;
  double  mat1[N1][N2];
  double  mat2[N1][N2];
  double  mat3[N1][N2];
  ...
  erg = summe2( mat2, n1, n2 );                 /* Aufruf der Funkt. summe2 */
  ...
}       /* ----------  end of function main  ---------- */
```

Die Übergabe zusätzlicher Parameter, hier n1 und n2, erlaubt, bei der Verwendung des Feldes innerhalb der Funktion nur einen Teilbereich zu verwenden.
Wenn es die Aufgabenstellung erlaubt, weil abzuschätzen ist, daß die auftretenden Felder beziehungsweise Datenstrukturen eine bestimmte Größe nicht überschreiten können, dann können diese sogenannten statischen Felder verwendet werden.

Mit dem Standard ***C99*** sind zwei Möglichkeiten der Feldübergabe hinzugekommen. Das nächste Beispiel zeigt ein sogenanntes offenes Feld. Die Feldgröße (n1×n2, 3. Parameter) wird beim Aufruf dieser Funktion durch die ersten beiden Parameter festgelegt. Aus diesem Grund ist die angegebene Reihenfolge der Parameter zwingend einzuhalten.

```
int f ( int n1, int n2, int feld[n1][n2] )
{
  ...
}       /* ----------  end of function f  ---------- */
```

Im folgenden Beispiel werden nur die Felddimensionen als Parameter übergeben. In der Funktion g wird dann ein Feld entsprechender Größe bei jedem Funktionsaufruf angelegt. Die Lebensdauer dieses Feldes ist allerdings auf die Funktion g begrenzt.

```
int g ( int n1, int n2 )
{
```

```
  int feld[n1][n2];
  ...
}       /* ----------  end of function g  ---------- */
```

6.7. Dynamische Speicherbelegung

In allen Fällen, in denen die bisher vorgestellten Möglichkeiten nicht passend sind, besteht die Möglichkeit, zur Laufzeit eines Programmes Speicher beim Betriebssystem anzufordern, zu verwenden und zu einem beliebigen Zeitpunkt wieder freizugeben. Die Anzahl und die Größe der angeforderten Blöcke ist nur durch den im verwendeten Rechner vorhandenen Speicher und die Größe des heaps im Speicherabbild des Prozesses (siehe Anhang A) begrenzt. Zur Speicherplatzbeschaffung und -freigabe stehen folgende Funktionen (`stdlib.h`) zur Verfügung:

`void *calloc(size_t nmemb, size_t size)`	Beschafft Speicherplatz für ein Feld mit `nmemb` Elementen von jeweils `size` Bytes Größe. Gibt im Erfolgsfall einen Zeiger auf den Bereichsanfang zurück. Die Elemente werden mit 0 initialisiert.
`void *malloc(size_t size)`	Beschafft `size` Bytes. Gibt im Erfolgsfall einen Zeiger auf den Bereichsanfang zurück. Der Speicherplatz wird nicht initialisiert.
`void free(void *ptr)`	Gibt den Speicherplatz frei, auf den der Zeiger `ptr` zeigt.

In Liste 6.3 wird die Verwendung dieser Funktionen an einem Beispiel gezeigt.

Liste 6.3: Dynamische Speicherbelegung mit `calloc`

```
1  #include  <stdio.h>
2  #include  <stdlib.h>
3
4  int main (void)
5  {
6    long    n;
7    double  *feld;
8
9    n    = 10000000L;                          /* Feldgröße           */
10
11   feld = calloc( n, sizeof(double) );        /* Speicher beschaffen */
12
13   if( feld == NULL ) {
14     printf("\nSpeicherplatzbeschaffung fehlgeschlagen !\n");
15     exit(0);                                  /* Programmabbruch     */
16   }
17
18   /* Verwendung des Feldes feld[] */
19
20   free(feld);                                 /* Speicher freigeben  */
21
22   return 0;
23 }       /* ----------  end of function main  ---------- */
```

In Zeile 11 wird die Größe eines Feldelementes mit Hilfe des Operators `sizeof` bestimmt.
In Zeile 13 wird überprüft, ob die Speicherplatzbeschaffung erfolgreich war. Konnte der angeforderte Speicher nicht zur Verfügung gestellt werden, dann erhält der Zeiger einen besonderen Rückgabewert, der dem vordefinierten Makro `NULL` entspricht. In diesem Falle wird das Programm mit einer Fehlermeldung abgebrochen.
In Zeile 20 wird der belegte Speicher durch die Funktion `free` wieder freigegeben.

Der mit `calloc` oder `malloc` dynamisch belegte Speicher bleibt bis zur Freigabe mit `free` belegt, längstens jedoch bis zum Ende des Programmlaufes. Wird unbeabsichtigt der Zeiger überschrieben der auf seinen Anfang zeigt, dann ist der Speicherplatz nicht mehr zugreifbar, bleibt aber belegt! Bei langlaufenden Programmen oder bei Programmen, die häufig Speicher belegen und freigeben, können solche Speicherverluste, auch Speicherlecks genannt, den gesamten verfügbaren Speicher aufzehren!

`void`-Zeiger

Die beiden Funktionen `calloc` oder `malloc` geben laut Definition einen Zeiger vom Typ `void` zurück. Derartige Zeiger sind generisch oder typ-neutral und müssen vor Gebrauch in andere Zeigertypen umgewandelt werden. Das geschieht zum Beispiel in der Zeile

```
feld = calloc( n, sizeof(double) ); /* Speicher beschaffen */
```

in der der von der Funktion `calloc` zurückgegebene `void`-Zeiger automatisch in einen `double`-Zeiger gewandelt wird (Liste 6.3).
Die Funktion `free` erwartet laut Definition einen `void`-Zeiger, erhält aber in der Zeile

```
free(feld);                          /* Speicher freigeben  */
```

einen `double`-Zeiger. Auch hier findet eine automatische Typumwandlung statt.
Speicherzugriffe durch Dereferenzierung von `void`-Zeigern sind nicht möglich. Mit Hilfe von `void`-Zeigern können aber generische Datenstrukturen aufgebaut werden (zum Beispiel Listen und Bäume), in die dann über Zeiger Daten unterschiedlichen Typs aufgenommen werden können.

Manuelle und automatische Speicherverwaltung

In ***C*** wird die Speicherverwaltung dem Programmierer aufgebürdet. Die Speicherverwaltung ist eine erhebliche Quelle von Fehlern, die oft nur mit Mühe und Aufwand gefunden und beseitigt werden können. Andere Programmiersprachen, insbesondere Skriptsprachen, bieten eine automatische Speicherverwaltung. Felder können einfach verlängert werden, Datenstrukturen, die nicht mehr benötigt werden, werden automatisch dem freien Speicher wieder hinzugefügt (automatische Speicherbereinigung, engl. *garbage collection*). Die Speicherverwaltung entfällt damit als Fehlerquelle. Die Speicherbereinigung startet jedoch unter Umständen zu einem ungünstigen und nicht vorhersehbaren Zeitpunkt während des Programmlaufes und benötigt für kurze Zeit (Sekundenbruchteile) Prozessorleistung. In einer dialogorientierten Anwendung ist das kein Problem. In Treibern, Betriebssystemteilen oder Echtzeitanwendungen ist das aber oft nicht hinnehmbar. Mit der manuellen Speicherverwaltung hat der Programmierer hier stets die Kontrolle über die Abläufe. Für zeitkritische Anwendungen stehen jedoch auch Lösungen mit inkrementeller oder präemptiver Speicherbereinigung zur Verfügung.

6.8. Zeiger und Zeichenketten

Liste 6.4: Kopieren und Wenden von Zeichenketten

```
1  #include  <errno.h>
2  #include  <math.h>
3  #include  <stdio.h>
4  #include  <stdlib.h>
5  #include  <string.h>
6
7  #define BUFFERLENGTH  64                        /* Länge char-Puffer */
8
9  /*--------------------------------------------------------------------------
10  *  string_copy
11  *-------------------------------------------------------------------------*/
12 void  string_copy ( char *copy, char *orig )
13 {
14   while ( ( *copy++ = *orig++ ) );
15 }   /* -----  end of function string_copy  ----- */
16
17 /*--------------------------------------------------------------------------
18  *  string_revers
19  *-------------------------------------------------------------------------*/
20 void string_revers  ( char *copy, char *orig  )
21 {
22   char  *po = orig + strlen(orig);              /* Endadresse Original      */
23   while ( --po >= orig  )
24     *copy++ = *po;
25   *copy = '\0';                                 /* Zeichenkette abschließen */
26 }   /* -----  end of function string_revers  ----- */
27
28 /*--------------------------------------------------------------------------
29  *  Hauptprogramm
30  *-------------------------------------------------------------------------*/
31 int main ( int argc, char *argv[] )
32 {
33   char  buffer[BUFFERLENGTH];                   /* char-Feld                */
34   char  *str1 = "abcdefgh";                     /* Zeichenkette fester Länge */
35   char  *str2 = malloc ( strlen(str1) + 1 );    /* dynamische Zeichenkette   */
36   if ( str2==NULL ) {
37     fprintf ( stderr, "\ndynamic memory allocation failed\n" );
38     exit (EXIT_FAILURE);
39   }
40   printf ( "str1   = '%s'\n", str1 );
41   string_copy( str2, str1 );                    /* Zeichenkette kopieren */
42   printf ( "str2   = '%s'\n", str2 );
43   string_revers( str2, str1 );                  /* Zeichenkette wenden   */
44   printf ( "str2   = '%s'\n", str2 );
45   string_revers( buffer, str1 );                /* Zeichenkette wenden   */
46   printf ( "buffer = '%s'\n", str2 );
47
48   free (str2);
49   return EXIT_SUCCESS;
50 }   /* ----------  end of function main  ---------- */
```

Eine wichtige Anwendung von Zeigern ist der Umgang mit Zeichenketten. Die Anweisung

```
char  *str1 = "abcdefgh";
```

vereinbart den Zeiger `str1`, der auf eine konstante Zeichenkette als Initialisierungswert zeigt. Die Verwendung eines Feldes fester Größe ist ebenfalls möglich:

```
char  buffer[BUFFERLENGTH];
```

Eine dynamische Zeichenkette wird mittels `malloc` zur Laufzeit angelegt, zum Beispiel:

```
char  *str2 = malloc ( strlen(str1) + 1 );
```

Liste 6.4 zeigt den Umgang mit diesen Zeichenketten. Die Funktion `string_copy` kopiert eine Zeichenkette auf eine andere. Dazu wird Adreßübergabe verwendet. Der Kopiervorgang reduziert sich auf eine leere Schleife:

```
while ( ( *copy++ = *orig++ ) );
```

Beginnend mit dem ersten Zeichen der Quelle, werden alle Zeichen durch Zuweisung auf die Zielzeichenkette kopiert. Nach jeder Zuweisung werden die beiden Zeiger inkrementiert. Wenn der Wert der inneren Klammer Null ist, weil das Endezeichen `'\0'` kopiert wurde, bricht die Schleife ab: der Wert der inneren Klammer ist dann Null, was als Wahrheitswert *falsch* interpretiert wird. Die Funktion `string_reverse` zum Wenden einer Zeichenkette ist geringfügig aufwendiger, weil die Quelle rückwärts gelesen wird, während das Ziel vorwärts zu schreiben ist.

In beiden Funktionen werden die übergebenen Zeiger durch Inkremetierung oder Dekrementierung verändert. Die Funktionen erhalten beim Aufruf Kopien der Adressen, die in den Zeigervariablen der aufrufenden Umgebung (hier `str1` und `str2` im Hauptprogramm) enthalten sind. Diese Kopien sind nur als lokale Größen innerhalb der Funktionen verfügbar und können deshalb verändert werden, ohne die Zeigervariablen in der aufrufenden Umgebung zu beeinflussen. Die Implementierung der beiden Funktionen ist *C*-typisch und etwas gewöhnungsbedürftig, aber effizient.

6.9. Funktionen als Funktionsargumente

In *C* können auch Zeiger auf Funktionen verwendet werden. Funktionszeiger können in entsprechenden Variablen oder Feldern gespeichert werden. Sie können aber auch als Parameter an Funktionen übergeben und als Rückgabewerte von Funktionen zurückgegeben werden. Mit Hilfe dieser Möglichkeiten lassen sich einige Aufgabenstellungen elegant lösen.

Soll eine Funktion an eine andere Funktion als Parameter übergeben werden, dann muß der formale Parameter in der Funktionsdefinition die zu übergebenden Funktion beschreiben. Die folgende Funktion `tabelliere_funktion` übernimmt als ersten Parameter einen Zeiger `f` auf eine Funktion, die selbst ein `double`-Argument hat und einen `double`-Wert zurückgibt:

```
void tabelliere_funktion (  double f(double),   /* Zeiger auf eine Funktion     */
                            double x_start,     /* erster x-Wert                */
                            double x_ende,      /* letzter x-Wert               */
                            int    n            /* Anzahl zu berechnender Werte */
                            )
{
  ...
}   /* -----  end of function tabelliere_funktion  ----- */
```

Die Funktion `tabelliere_funktion` soll eine Tabelle von Funktionswerten ausgeben. Die zu tabellierende Funktion wird als Parameter übergeben. Der erste Parameter kann auch in Zeigerschreibweise geschrieben werden:

```
double (*f)(double)
```

Die Klammern um `*f` sind notwendig. Die Parameterangabe

```
double *f(double)
```

beschreibt eine Funktion, die einen Zeiger auf eine `double`-Größe zurückgibt. Die Funktion `sin` aus der *C*-Standardbibliothek entspricht genau der Zeigerbeschreibung `double f(double)`. Ein Aufruf der Funktion `tabelliere_funktion` sieht deshalb zum Beispiel wie folgt aussehen:

```
tabelliere_funktion ( sin, x0, x1, n );
```

Liste 6.5 zeigt ein vollständiges Beispielprogramm, in dem zwei unterschiedliche Funktionen über Zeiger aufgerufen werden.

Liste 6.5: Verwendung einer Funktion als Funktionsargument

```
1  #include  <math.h>
2  #include  <stdio.h>
3
4  /*-------------------------------------------------------------------------
5   *  tabelliere_funktion : Tabelle mit Funktionswerten ausgeben
6   *-------------------------------------------------------------------------*/
7  void tabelliere_funktion (  double f(double), /* Zeiger auf eine Funktion     */
8                              double x_start,   /* erster x-Wert                */
9                              double x_ende,    /* letzter x-Wert               */
10                             int    n          /* Anzahl zu berechnender Werte */
11                             )
12 {
13   int i;
14   double  x;
15   printf ("\n Punkt             x          f(x)");
16   printf ("\n ---------------------------------\n");
17   for ( i = 0; i < n; i += 1 ) {
18     x = i*(x_ende-x_start)/(n-1);
19     printf ("  %4d    %10.4lf    %10.4lf\n", i+1, x, f(x) );
20   }
21 }   /* -----  end of function tabelliere_funktion  ----- */
22
23 /*-------------------------------------------------------------------------
24  *  polynom3 : Polynom 3. Ordnung y(x) = 12 + 2*x + 3*x^2 + x^3
25  *-------------------------------------------------------------------------*/
26 double polynom3 ( double x )
27 {
28   return 12 + x*(2 + x*(3 + x));
29 }   /* -----  end of function polynom3  ----- */
30
31 int main ( void )              /* -------------- Hauptprogramm -------------- */
32 {
33   int     n   = 9;             /* Anzahl der zu tabellierenden Funktionswerte */
34   double  x0  = 0.0;           /* erster x-Wert                               */
35   double  x1  = 2.0;           /* letzter x-Wert                              */
36
37   tabelliere_funktion ( polynom3, x0, x1, n );           /* Tabelle ausgeben */
```

```
38   tabelliere_funktion ( sin,      x0, x1, n );           /* Tabelle ausgeben */
39
40   return 0;
41 }   /* ----------  end of function main  ---------- */
```

6.10. Generische Programmierung in *C*

Sortieralgorithmen wie Quicksort oder Bubblesort, aber auch andere Algorithmen, lassen sich oft weitgehend unabhängig von einem speziellen Datentyp beschreiben. Die Abhängigkeit vom Typ der zu sortierenden Daten ist nur an den Stellen des Algorithmus gegeben, an denen zwei Werte verglichen werden müßen, um deren Anordnung zu ändern. Beim Sortieren von Zahlen wird dazu ein Größenvergleich vorgenommen. Die Sortierung einer Bildfolge oder einer Menge von Zeichenketten erfordert dagegen den Vergleich von völlig anderen Merkmalen.
Eine Idee bei der generischen oder typfreien Programmierung ist es, einen Algorithmus getrennt von seinen datentypabhängigen Anteilen zu implementieren. Wird der Algorithmus in einer anderen Aufgabenstellung erneut verwendet, dann müssen nur noch die datentypabhängigen Anteile neu erstellt werden.

Da in *C* aber immer Datentypen verwendet werden müssen, kann folgende Vorgehensweise gewählt werden:

- Der generische Anteil einer Aufgabe wird mit Hilfe von `void`-Zeigern programmiert.
- Der datentypabhängige Anteil wird wie üblich für den zu verwendeten Zieldatentyp programmiert.
- Da `void`-Zeiger nicht dereferenziert werden können, müssen beide Teile durch Zeigertypumwandlung aneinander angepaßt werden.

Die Funktion `qsort` der *C*-Standardbibliothek (`stdlib.h`) ist ein Beispiel für eine generische Sortierfunktion. Sie sortiert ein Feld. Der Prototyp sieht wie folgt aus:

```
void qsort ( void   *base,
             size_t  nmemb,
             size_t  size,
             int     (*compar)(const void *, const void *)
           );
```

Die Parameter haben folgende Bedeutung:

`void *base`	Zeiger auf den Feldanfang
`size_t nmemb`	Anzahl der Feldelemente
`size_t size`	Größe eines Feldelementes in Byte
`int (*compar)(const void *, const void *)`	Vergleichsfunktion für zwei Feldelemente

Bei einem typisierten Feld wird der Adreßabstand der Feldelemente vom Compiler aus dem Datentyp abgeleitet (zum Beispiel 4 Byte beim Datentyp `int` mit einer 32-Bit-Darstellung). Da das bei einem `void`-Zeiger so nicht möglich ist, erhält die Funktion `qsort` die Größe der Feldelemente als dritten Parameter.
Der vierte Parameter ist ein Zeiger auf eine Vergleichsfunktion. Diese ist natürlich vom tatsächlichen Datentyp des Feldes abhängig und muß deshalb vom Anwender jeweils getrennt erstellt werden. Sie liefert einen Rückgabewert kleiner, gleich oder größer Null, je nachdem ob

ihr erstes Argument kleiner, gleich oder größer als ihr zweites Argument ist.
Liste 6.6 zeigt als Beispiel die Sortierung eines `double`-Feldes. Die Schnittstelle der Funktion `compare_double` entspricht den Vorgaben. Im Inneren müssen die `void`-Zeiger vor dem Vergleich in `double`-Zeigern umgewandelt werden:

```
if ( *(double*)p1 < *(double*)p2 )
```

Erst danach kann durch Dereferenzierung auf die Inhalte zugegriffen werden, die für den numerischen Vergleich erforderlich sind.

Liste 6.6: Verwendung der Funktion `qsort`

```
1  #include  <stdlib.h>
2
3  /*
4   * ===  FUNCTION  ======================================================
5   *         Name:  compare_double
6   *  Description:  Vergleich zweier double-Größen
7   *                Rückgabewerte:
8   *                -1  :  Argument1  < Argument2
9   *                 0  :  Argument1 == Argument2
10  *                +1  :  Argument1  > Argument2
11  * =====================================================================
12  */
13 int compare_double ( const void *p1, const void *p2 )
14 {
15   if ( *(double*)p1 < *(double*)p2 )   /* Typumwandlung: (void*) => (double*) */
16     return -1;
17
18   if ( *(double*)p1 > *(double*)p2 )   /* Typumwandlung: (void*) => (double*) */
19     return +1;
20
21   return 0;
22 }   /* -----  end of function compare_double  ----- */
23
24 /*-------------------------------------------------------------------------
25  *  Hauptprogramm
26  *-------------------------------------------------------------------------*/
27 int main ( void )
28 {
29   double  array[] = { 12, 2, 12, 12, -3, 33, 44, 333, 99 };
30
31   size_t  size  = sizeof( double );             /* Größe eines Feldelem. (Byte)*/
32   size_t  nmemb = sizeof( array )/size;         /* Anzahl der Feldelemente     */
33
34   qsort ( array, nmemb, size, compare_double ); /* Feldsortierung              */
35
36   return EXIT_SUCCESS;
37 }   /* ----------  end of function main  ---------- */
```

Im Hauptprogramm in Liste 6.6 wird beim Aufruf von `qsort` der Feldname `array` als Zeiger übergeben:

```
qsort ( array, nmemb, size, compare_double );
```

Da die Funktion `qsort` als ersten Parameter einen `void`-Zeiger erwartet, findet bei der Parameterübergabe eine implizite Typumwandlung zu einem `void`-Zeiger statt.

7. Strukturen

Die Bezeichnung *Datenstrukturen* ist im allgemeinen ein Oberbegriff für viele Arten von planmäßigen Zusammenstellungen und Anordnungen unterschiedlichster Daten. Der Begriff ist in diesem Sinne nicht an eine bestimmte Programmiersprache gebunden.
Ein Feld von `double`-Werten ist zum Beispiel eine Datenstruktur, die viele *gleichartige* Elemente eines Grunddatentyps in einer geordneten Weise und über Indizierung zugreifbar zur Verfügung stellt.
Darüber hinaus gibt es in vielen Programmiersprachen die Möglichkeit, Datenelement mit *unterschiedlichen Datentypen* zu einer Einheit, nämlich einer Struktur im engeren Sinne, zusammenzufassen. In ***C*** werden derartige Strukturen mit Hilfe des Schlüsselwortes `struct` vereinbart. Sie stellen einen benutzerdefinierten Datentyp dar.

7.1. Definition und Komponentenzugriff

Die Vereinbarung einer Struktur hat folgende allgemeine Form:

```
struct  Name {
  Komponente 1
  Komponente 2
   ...
};
```

Dem Schlüsselwort `struct` folgt ein frei gewählter Name für die neue Struktur. Beide Bezeichnungen zusammen bilden die Bezeichnung für den neuen Datentyp. Die Komponenten sind Vereinbarungen von Größen, deren Datentyp bereits bekannt ist.
Eine kleine Datenstruktur zur Verwaltung von Studentendaten könnte zum Beispiel wie folgt vereinbart werden:

```
struct Student {
  char  nachname[30];
  char  vorname[30];
  long  matrikelnummer;
  int   studiengang;
};
```

Damit ist der neue Datentyp `struct Student` eingerichtet. Zur Vereinbarung von Variablen diesen Typs wird wie üblich verfahren: hinter der Datentypbezeichnung folgt eine Liste von Variablen-, Feld- oder Zeigernamen, die diesen Typ besitzen sollen.

```
struct Student   std1, std2;                  /* 2 Einzelvariablen        */
struct Student   studenten[8000];             /* Feld aus Strukturen      */
struct Student  *pstd;                        /* Zeiger auf eine Struktur */
```

Die Strukturvereinbarung selbst ist nur eine Schablone. Speicherplatz wird nur angelegt, wenn auch tatsächlich eine Struktur vereinbart wird.

Der Zugriff auf eine Komponente einer Struktur erfolgt mit Hilfe des Punktoperators und des Komponentennamen. Eine der Komponenten der Struktur `Student` heißt `matrikelnummer`. Die Zuweisung einer Matrikelnummer an diese Strukturkomponente geschieht wie folgt:

```
std1.matrikelnummer = 10004711;

printf ("std1.matrikelnummer = %ld\n", std1.matrikelnummer );

studenten[5]  = std1;                        /* Zuweisung an ein Feldelement */

pstd  = &std1;                               /* Adresse übernehmen           */
```

Die Komponente `matrikelnummer` ist definitionsgemäß eine `long`-Größe und wird genauso behandelt. Das gilt unter anderem auch für die Ein-/Ausgabe.
Strukturen können einander vollständig zugewiesen werden, wie die Zuweisung von `std1` an ein Feldelement zeigt.
Adressen auf Strukturen werden in der üblichen Weise mit Hilfe des Adreßoperators gewonnen.

Die Verwendung von `typedef`

Die Verwendung der in ***C*** möglichen, mehrteiligen Typbezeichnungen, wie zum Beispiel `const unsigned long int` oder `struct Student`, wird oft als unbequem und schlecht lesbar empfunden. Mit Hilfe einer `typedef`-Anweisung kann einer bereits definierten Typbezeichnung ein neuer Name zugeordnet werden. Die allgemeine Form lautet:

`typedef`	*existierender Datentyp*	***Name*****`;`**

Hierzu einige Beispiele:

```
typedef  const unsigned long int    culi;
typedef  unsigned char              byte;
typedef  struct Student             Student;

typedef  double                     skalar;
typedef  skalar                     vektor[3];
typedef  skalar                     matrix[3][3];
```

Die `typedef`-Anweisung kann offensichtlich auch dazu verwendet werden, problembezogene Datentypnamen (wie etwa `skalar` oder `vektor`) festzulegen. Mit der Festlegung von Namen für Felder (`vektor`,`matrix`) kann bei entsprechenden Vereinbarungen die Angabe der Feldgrößen unterbleiben. Die folgenden Zeilen zeigen die Anwendung:

```
byte    b1, b2, b3;
skalar  s1, s2;
vektor  u, v, w;          /* Felder, Länge 3 */
matrix  m1, m2;           /* 3x3-Matrizen    */

b1  = b2 = b3 = 'c';
s1  = 23.3;
s2  = 11.9;

u[0]  = +1.0;
u[1]  =  0.0;
```

```
u[2]  = -1.0;

m1[0][0]  = m1[1][1]  = m1[2][2]  = 1.0;
```

Liste 7.1 zeigt die Darstellung von Polarkoordinaten als Struktur. Die beiden Strukturelement besitzen hier denselben Datentyp. Die Verwendung einer Struktur ist trotzdem vorteilhaft, weil die Elemente mit ihren Namen angesprochen werden können und damit einen Beitrag zur Lesbarkeit des Programmes leisten.

Liste 7.1: Polarkoordinaten als Struktur

```
1
2  #include  <stdio.h>
3
4  struct polarkoordinaten
5  {
6    double  rad;                                  /* Radius in Millimeter */
7    double  phi;                                  /* Winkel im Bogenmaß   */
8  };
9
10 typedef struct polarkoordinaten polarkoordinaten;
11
12
13 int
14 main ( int argc, char *argv[] )
15 {
16   polarkoordinaten  p1, p2;
17   polarkoordinaten  p3 = { 1.0, 0.0 };          /* Initialisierung   */
18
19   p1.rad = 27.0;                                /* Radius zuweisen   */
20   p1.phi =  0.3;                                /* Winkel zuweisen   */
21   p2     = p1;                                  /* Struktur zuweisen */
22
23   printf ("p1 = ( %10.2f , %10.2f )\n", p1.rad, p1.phi );
24   printf ("p2 = ( %10.2f , %10.2f )\n", p2.rad, p2.phi );
25   printf ("p3 = ( %10.2f , %10.2f )\n", p3.rad, p3.phi );
26
27   return 0;
28 }       /* ----------  end of function main  ---------- */
```

7.2. Strukturen und Funktionen

Strukturen können als Parameter an Funktionen übergeben und über `return` auch aus Funktionen zurückgegeben werden.

Kleine Strukturen werden meist durch Wertübergabe an Funktionen übergeben. In diesem Fall werden Kopien aller Bestandteile übergeben (für das Beispiel der Polarkoordinaten in Liste 7.1 also stets zwei `double`-Größen).
Große Strukturen werden in der Regel aus Aufwandsgründen durch Adreßübergabe an Funktionen übergeben. Liste 7.2 zeigt ein einfaches Beispiel.

Liste 7.2: Adreßübergabe bei Strukturen

```
1  #include  <stdio.h>
2
3  struct student  { char  nachname [31];      /* Nachname, max. 30 Zeichen  */
4                    char  vorname  [31];      /* Vorname, max. 30 Zeichen   */
5                    long  matr_nr;            /* Matrikelnummer             */
6                    int   studiengang;        /* Kennzahl des Studienganges */
7                    };
8
9  typedef struct student  Student;
10
11 void  lies_student    ( Student *s )
12 {
13   printf("\n\n\tNeuer Datensatz Student");
14   printf("\n\nNachname    : "); scanf( "%s",  s->nachname    );
15   printf(  "\nVorname     : "); scanf( "%s",  s->vorname     );
16   printf(  "\nMatrikelnr. : "); scanf("%ld", &s->matr_nr     );
17   printf(  "\nStudiengang : "); scanf( "%d", &s->studiengang );
18 }
19
20 void print_student ( Student *s )
21 {
22   printf("\nNachname:%30s, Vorname:%30s\n"    , (*s).nachname, (*s).vorname     );
23   printf(  "Matr.Nr.:%8ld, Studiengang: %4d\n", (*s).matr_nr , (*s).studiengang );
24 }
25
26 int main ( int argc, char *argv[] )
27 {
28   Student stud[1000];
29
30   lies_student  ( &stud[23] );
31   print_student ( &stud[23] );
32
33   return 0;
34 }       /* ----------  end of function main  ---------- */
```

Die Struktur **struct** student erhält zunächst mittels typedef den Namen Student. Die Funktion lies_student liest in einem einfachen Dialog die vier Komponenten einer Struktur ein. Hierzu muß die Adresse einer Struktur als Parameter übergeben werden.
Die Funktion print_student gibt eine derartige Struktur aus. Diese Funktion erhält ebenfalls einen Zeiger auf eine Struktur.
Im Hauptprogramm wird ein Feld von 1000 Strukturen des Typs Student angelegt und lediglich eine Struktur in das Feldelement mit dem Index 23 eingelesen. Der Parameter der beiden Funktionsaufrufe lautet &stud[23]. Er beschafft die Adresse der Struktur mit dem Index 23.

Im Innern der beiden Funktionen werden zwei unterschiedliche Schreibweisen für den Zugriff auf die Strukturkomponenten über die übergebenen Zeiger verwendet. In der Funktion print_student wird die Komponente nachname wie folgt angesprochen:

```
(*s).nachname
```

Die Dereferenzierung des Zeigers durch (*s) liefert zunächst die adressierte Struktur. Mit Hilfe des Punktoperators kann nun auf die Komponente nachname zugegriffen werden. Die Klammerung ist hier zwingend erforderlich, weil der Punktoperator einen höheren Vorrang besitzt als der Inhaltsoperator.

Weil diese Schreibweise unschön und umständlich ist, wurde eine zusätzliche Schreibweise für den Zugriff auf Strukturkomponenten über Zeiger eingeführt. In der Funktion `lies_student` wird die Komponente `nachname` durch

```
s->nachname
```

angesprochen. Der Zugriff über den Pfeiloperator `->` vereinigt die Dereferenzierung und den Zugriff über den Punktoperator.

Die Listen 7.3 und 7.4 zeigen ein etwas umfangreicheres Beispiel für die Verwendung einer einfachen Struktur. Die Struktur `struct vek3` faßt drei `double`-Größen zu einem Vektor mit den Komponenten `x`, `y` und `z` zusammen. Innerhalb der Funktionen werden dadurch die Vektorkomponenten mit ihren in der Mathematik gebräuchlichen Namen ansprechbar.
Die Funktionen verwenden durchgängig Wertübergabe und Wertrückgabe. Sie können dadurch ihrerseits als Funktionsparameter verwendet werden, um zusammengesetzte Operationen aus einfacheren erzeugen zu können. So kann zum Beispiel ein mit dem Faktor 0.75 gestauchter Vektor in y-Richtung einfach wie folgt erzeugt und zugewiesen werden:

```
b = vxs ( e_y(), 0.75 );                    /* 3/4 * y-Einheitsvektor */
```

Liste 7.3: Darstellung von Vektoren der Länge 3 mittels Strukturen

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

#define   EPSILON   1.0E-10   /* kürzere Vektoren werden als */
                              /* Nullvektor behandelt        */

/* ----------  struct vek3 : Vektoren der Länge 3  ---------- */
struct vek3 { double  x, y, z; };

typedef struct vek3 vek3;

/* ---------- createv : Initialisierung ---------- */
vek3 createv ( double x, double y, double z ) {
  vek3  erg = { x, y, z };
  return erg;
}

/* ---------- Nullvektor, Einheitsvektoren ---------- */
vek3 null (  ) { return createv( 0.0, 0.0, 0.0 ); }
vek3 e_x  (  ) { return createv( 1.0, 0.0, 0.0 ); }
vek3 e_y  (  ) { return createv( 0.0, 1.0, 0.0 ); }
vek3 e_z  (  ) { return createv( 0.0, 0.0, 1.0 ); }

/* ---------- vxs : Vektor * Skalar ---------- */
vek3 vxs ( vek3 v, double s ) {
  return createv( s*v.x, s*v.y, s*v.z );
}

/* ---------- vaddv : Vektoraddition ---------- */
vek3 vaddv ( vek3 a, vek3 b ) {
  return createv( a.x+b.x, a.y+b.y, a.z+b.z );
}

/* ---------- vsubv : Vektorsubtraktion ---------- */
vek3 vsubv ( vek3 a, vek3 b ) {
  return createv( a.x-b.x, a.y-b.y, a.z-b.z );
}

/* ---------- Skalarprodukt ---------- */
double skalarprodukt ( vek3 a, vek3 b ) {
  return a.x*b.x + a.y*b.y + a.z*b.z;
}

/* ---------- Kreuzprodukt ---------- */
vek3 kreuzprodukt ( vek3 a, vek3 b ) {
  return  createv ( a.y*b.z - a.z*b.y,
                    a.z*b.x - a.x*b.z,
                    a.x*b.y - a.y*b.x );
}

/* ---------- laenge : Vektorlänge ---------- */
double laenge ( vek3 a ) {
  return sqrt(skalarprodukt(a,a));
}
```

Liste 7.4: Darstellung von Vektoren der Länge 3 mittels Strukturen (Fortsetzung)

```
55 /* ---------- normieren : Vektor normieren ---------- */
56 vek3  normieren ( vek3 a ) {
57   double  l = laenge(a);
58   if (l>EPSILON)                        /* Division durch Null vermeiden */
59     return a;
60   return vxs( a, 1.0/l );
61 }
62
63 /* ---------- Einfache Ausgabe ---------- */
64 void print_vek3 ( vek3 a ) {
65   printf("[ %.4g , %.4g , %.4g ]", a.x, a.y, a.z );
66 }
67
68 /* ---------- Hauptprogramm ---------- */
69 int main ( void )
70 {
71   vek3  a, b, c;
72
73   a = createv( 1, 2, 3 );
74   b = createv( 3, 1, 2 );
75   c = vaddv ( a, b );                              /* Vektor c = a + b       */
76
77   printf("\n\nVektoraddition  c = a + b");
78   printf("\nVektor a = ");  print_vek3( a );      /* Vektor a ausgeben      */
79   printf("\nVektor b = ");  print_vek3( b );      /* Vektor b ausgeben      */
80   printf("\nVektor c = ");  print_vek3( c );      /* Vektor c ausgeben      */
81
82   c = vxs ( a, 0.11 );                             /* Skalarmultiplikation   */
83
84   printf("\n\nSkalarmultiplikation  c = 0.11*a");
85   printf("\nVektor a = ");  print_vek3( a );      /* Vektor a ausgeben      */
86   printf("\nVektor c = ");  print_vek3( c );      /* Vektor c ausgeben      */
87
88   c = normieren(a);                                /* Vektor normieren       */
89
90   printf("\n\nVektor normieren  c = a_normiert");
91   printf("\nVektor a = ");  print_vek3( a );      /* Vektor a ausgeben      */
92   printf("\nVektor c = ");  print_vek3( c );      /* Vektor c ausgeben      */
93   printf("\nLänge  c = %f (Kontrolle)", laenge(c));
94
95   printf("\n\nKreuzprodukt c = a x b\n");
96   a = vxs ( e_x(), 0.25 );                         /* 1/4 * x-Einheitsvektor */
97   b = vxs ( e_y(), 0.75 );                         /* 3/4 * y-Einheitsvektor */
98   c = kreuzprodukt(a,b);                           /* Kreuzprodukt           */
99
100  printf("\nVektor a = ");  print_vek3( a );
101  printf("\nVektor b = ");  print_vek3( b );
102  printf("\nVektor c = ");  print_vek3( c );      /* Vektor c ausgeben      */
103  return 0;
104 }       /* ----------  end of function main  ---------- */
```

8. Bit-Operationen und Aufzählungstypen

8.1. Bit-Operationen

Operator	Bedeutung
~	bitweises Komplement
<< >>	bitweise nach links, rechts schieben
&	bitweise UND
^	bitweise exklusives ODER
|	bitweise ODER
&&	logisches UND
||	logisches ODER
&= ^= |= <<= >>=	Verknüpfungen mit Zuweisung

Tabelle 8.1.: Bit-Operatoren, geordnet nach fallendem Vorrang

Bit-Operatoren (Tabelle 8.1) werden auf ganzzahlige Werte und Ausdrücke angewendet, die in diesem Falle als Folge einzelner Bits betrachtet werden. Da bei ganzzahligen Werten stets die interne Darstellung eine Rolle spielt weil dadurch die Anzahl der Bits des jeweiligen ganzzahligen Datentyps festgelegt ist (Abschnitt 2.3), sind diese Operatoren systemabhängig. Deshalb beschränkt sich die folgende Darstellung auf Bytes mit acht Bits und Ganzzahlen (Typ `integer`) mit vier Bytes.

Bitweises Komplement

00000000 00000000 00000011 11111101
11111111 11111111 11111100 00000010

Abbildung 8.1.: Interne Darstellung des ganzzahligen Wertes 1021 (32 Bits, oben) und des zugehörigen bitweisen Komplements (unten)

Der Operator ~ bildet das bitweise Komplement einer Größe. Nach der Definition und Wertzuweisung

```
int a = 1021;
```

besitzt die Variable `a` die in Abbildung 8.1 oben dargestellte, interne Bitfolge. Im Wert des Ausdrucks `~a` sind alle Bits invertiert (Abbildung 8.1, unten).

Und, Oder, Exklusives Oder

a	b	a&b	a|b	a^b
0	0	0	0	0
1	0	0	1	1
0	1	0	1	1
1	1	1	1	0

Tabelle 8.2.: Wirkungsweise der Operationen `&`, `|` und `^`

Zur bitweisen Und- und Oder-Verknüpfung stehen die Operatoren `&` (Und), `|` (Oder) und `^` (exklusives Oder) zur Verfügung. Tabelle 8.2 zeigt die Verküpfungsregeln, Abbildung 8.3 zeigt Beispiele mit konkreten Werten.

Ausdruck	**interne Darstellung**			
a	00000000	00000000	00000011	11111101
b	00000000	00000000	00000010	01010110
a&b	00000000	00000000	00000010	01010100
a|b	00000000	00000000	00000011	11111111
a^b	00000000	00000000	00000001	10101011

Tabelle 8.3.: Beispiele für die Anwendung von Bit-Operatoren

Schiebeoperatoren

Ausdruck	**interne Darstellung**			
a	00000000	00000000	00000011	11111101
a<<1	00000000	00000000	00000111	11111010
a<<2	00000000	00000000	00001111	11110100
a<<3	00000000	00000000	00011111	11101000
a<<31	10000000	00000000	00000000	00000000
a>>1	00000000	00000000	00000001	11111110
a>>2	00000000	00000000	00000000	11111111
a>>3	00000000	00000000	00000000	01111111

Tabelle 8.4.: Beispiele für die Anwendung der Schiebeoperatoren

Die allgemeine Form einer Linksschiebe-Operation `<<` lautet:

$$Ausdruck_1 \quad << \quad Ausdruck_2$$

Die beiden Ausdrücke müssen ganzzahlig sein. Die Bitfolge von $Ausdruck_1$ wird um die Anzahl von Bitpositionen nach links geschoben, die der $Ausdruck_2$ angibt. Dabei werden von

rechts Nullen nachgezogen. Für die Verschiebung der Bitfolge nach rechts (Operator >>) gilt sinngemäß das Gleiche. Tabelle 8.4 zeigt dazu einige Beispiele.

Liste 8.1: Ausgabe der Bit-Darstellung eines `int`-Wertes (nach [KP97])

```
1   void
2 bit_print_int ( unsigned int wert )
3 {
4   unsigned int  n     = sizeof(int)*CHAR_BIT; /* in limits.h                */
5   unsigned int  maske = 1U<<(n-1);           /* Maske 100 ... 000          */
6   unsigned int  i;                           /* Schleifenzähler            */
7
8   for ( i=1; i<=n; i++ ) {
9     if ( ( wert & maske ) == 0 )             /* höchstwertiges Bit ermitteln */
10      putchar('0');
11    else
12      putchar('1');
13    if ( i%CHAR_BIT == 0 ) putchar(' ');     /* Trennzeichen ausgegeben    */
14    wert  <<= 1;
15  }
16 }   /* -----  end of function bit_print_int  ----- */
```

Liste 8.1 zeigt ein Verwendungsbeispiel für Bit-Operationen. Die Funktion `bit_print_int` gibt die Bit-Darstellung eines `int`-Wertes aus. Die Anzahl der Bits der internen Darstellung wird in Zeile 4 aus der Anzahl der Bytes (`sizeof(int)`) und der Anzahl der Bits pro Byte (Makro `CHAR_BIT` aus der header-Datei `limits.h`) bestimmt. Dem folgenden Beispiel liegen 32 Bits zugrunde. Für den Wert 263 wird demnach

```
00000000 00000000 00000001 00000111
```

ausgeben. Die Idee ist dabei, die Bits von links nach rechts aufzugreifen. In Zeile 5 wird dazu eine sogenannte Maske definiert, die die Bitfolge

```
10000000 00000000 00000000 00000000
```

besitzt, also 32 Bits, bei denen nur das höchstwertige Bit den Wert 1 hat. Die Maske wird durch die Verschiebung des Wertes 1 um 31 Positionen nach links erzeugt und hat damit den numerischen Wert 2^{31}.

Die nachfolgende Schleife sorgt für die Behandlung aller 32 Bits. Der zu untersuchende Wert in `wert` und die Maske `maske` werden durch ein bitweises Und verknüpft, um das jeweils höchstwertige Bit herauszugreifen (Zeile 9):

```
( wert & maske )
```

Ist der Wert des Ausdrucks 0, dann ist auch das höchstwertige Bit 0, andernfalls hat es den Wert 1. Für den nächsten Schleifendurchlauf wird der Wert in `wert` um eine Position nach links gesetzt (Zeile 14):

```
wert <<= 1;
```

Zur Verbesserung der Lesbarkeit wird nach jeweils acht Bits ein Leerzeichen eingesetzt (Zeile 13). Mit Hilfe der Funktion `bit_print_int` sind die Beipiele dieses Abschnitts leicht nachzuvollziehen.

Bit-Masken

In Anwendungen werden gelegentlich Zustände, Rechte, Betriebsmodi oder Fehlercodes als Bitmaske codiert, wenn die einzelnen Informationen als Bit dargestellt werden können.
Eine Datei kann zum Beispiel gelesen, geschrieben oder ausgeführt werden. Die drei Rechte können in beliebiger Kombination auftreten und schließen sich nicht gegenseitig aus. Eine Betriebssystemunterscheidung mittels *win16*, *win32*, *win64*, *win95*, *unix*, *macunix*, *win32unix* läßt sich in sieben Bits codieren, die sich gegenseitig ausschließen.

Liste 8.2 zeigt ein weiteres Beispiel. Mittels `define` werden Masken `ROLE_xxx` definiert, in denen genau ein Bit gesetzt ist (Zweierpotenzen). Der Datentyp `Flags` dient ebenfalls zur Aufnahme von Bitmasken. Zur Initialisierung verwendet man Oktal- oder Hexadezimalzahlen, damit die gesetzten Bits leichter identifizierbar sind. Die oben dargestellten Bit-Operationen erlauben nun den Zugriff auf einzelne Bits oder auf Bit-Gruppen.

Liste 8.2: Verwendung von Bit-Masken zur Handhabung von Benutzerrollen

```
1  #include  <stdio.h>
2  #include  <stdlib.h>
3
4  #define ROLE_NONE        0      /* keine Rolle zugeordnet */
5  #define ROLE_GUEST       1      /* Gast                   */
6  #define ROLE_USER        2      /* normaler Benutzer      */
7  #define ROLE_NETADMIN    4      /* Netzwerk-Administrator */
8  #define ROLE_WEBADMIN    8      /* Web-Administrator      */
9  #define ROLE_SYSADMIN   16      /* System-Administrator   */
10
11 typedef unsigned int  Flags;
12
13   int
14 main ( int argc, char *argv[] )
15 {
16   Flags mask1 = 0x0C;                          /* Net-Admin oder Web-Admin */
17   Flags mask2 = 0x0E;                          /* unter anderem Benutzer   */
18   Flags admin = ROLE_NETADMIN | ROLE_WEBADMIN | ROLE_SYSADMIN;
19
20   if ( mask1 & ROLE_WEBADMIN ) {               /* Web-Administrator ?      */
21     printf ( "You are web-administrator.\n" );
22   }
23
24   if ( mask2 & admin ) {                       /* einer von 3 Admin-Typen? */
25     printf ( "You are an administrator.\n" );
26   }
27
28   if ( mask2 & ROLE_USER ) {                   /* normaler Benutzer?       */
29     mask2 ^= ROLE_USER;                        /* USER-Bit kippen          */
30     printf ( "You are no longer user!\n" );
31   }
32
33   return EXIT_SUCCESS;
34 }   /* ----------  end of function main  ---------- */
```

8.2. Aufzählungstypen

Das Schlüsselwort `enum` (engl. *enumeration* = Aufzählung) legt eine feste Menge von Bezeichnern mit zugeordneten Werten fest. Die Vereinbarung

```
enum monat {
  Jan, Feb, Mar, Apr, Mai, Jun, Jul, Aug, Sep, Okt, Nov, Dez
};
```

erzeugt den benutzerdefinierten Datentyp `enum monat`. Der Name `monat` ist frei gewählt. Die zwölf Monatskürzel sind ganzzahlige Konstanten mit aufsteigenden Werten von 0 (`Jan`) bis 11 (`Dez`). Gerade bei Monatsnamen ist der Beginn bei 0 unpraktisch. Die Vereinbarung wird deshalb durch die Zuweisung eines Startwertes ergänzt:

```
enum monat {
  Jan = 1, Feb, Mar, Apr, Mai, Jun, Jul, Aug, Sep, Okt, Nov, Dez
};

typedef enum monat Monat;
```

Außerdem wird mittels `typedef` die kürzere Typbezeichnung `Monat` eingeführt. Damit ist es möglich, Variablen vom Typ `enum monat` oder `Monat` einzurichten und zu verwenden:

```
enum monat  m1 = Jan;
Monat       mx;
  ...
if ( mx == m1 ) {
  ...
}

switch ( mx ) {
  case Jan:
    ...
    break;

  case Feb:
    ...
    break;

  ....
}       /* -----  end switch  ----- */
```

Aufzählungen sind äußerst nützlich, weil sie eine Menge zusammengehöriger, benannter Konstanten einführen, die die Lesbarkeit und Selbstdokumentation eines Programmes deutlich verbessern. Das folgende Beispiel aus einer Automatisierungslösung verdeutlicht das:

```
enum JointType {
  Fixed = 1, Revolute, Prismatic
};

struct joint {
  enum JointType  joint_type;
  double          position_min;       /* minimal position [mm] or [rad] */
  double          position_act;       /* actual  position [mm] or [rad] */
  double          position_max;       /* maximal position [mm] or [rad] */
};
```

```
struct joint glk[6]  = { { Prismatic,          0 , 200,         520  },
                         { Revolute,  deg2rad(-70),   0, deg2rad(+80) },
                         { Revolute,  deg2rad(-90),   0, deg2rad(+90) },
                           ...
};
```

Zur Klassifizierung von Gelenkverbindungen (engl. *joints*) stehen die Gelenktypen Festgelenk, Drehgelenk und Schubgelenk zur Verfügung (Typ `enum JointType`). Ein Gelenk wird durch eine Struktur vom Typ `struct joint` beschrieben, die den Gelenktyp, die untere Grenzlage, die aktuelle Lage und die obere Grenzlage enthält. Die erforderlichen sechs Gelenke werden als Feld von Strukturen mit Initialisierungsliste eingerichtet. Die Funktion `deg2rad` wandelt Gradangabe in Bogenmaß um.

Die Handhabung von Benutzerrollen in Liste 8.2 bietet ebenfalls eine Einsatzmöglichkeit für eine Aufzählungen. Die folgende Aufzählung `enum Flags` ersetzt die entsprechenden `define`-Größen. Der Rest des Programmes bleibt gleich.

```
enum Flags {
  ROLE_NONE     =  0,       /* keine Rolle zugeordnet */
  ROLE_GUEST    =  1,       /* Gast                   */
  ROLE_USER     =  2,       /* normaler Benutzer      */
  ROLE_NETADMIN =  4,       /* Netzwerk-Administrator */
  ROLE_WEBADMIN =  8,       /* Web-Administrator      */
  ROLE_SYSADMIN = 16,       /* System-Administrator   */
};        /* ----------  end of enum Flags  ---------- */

typedef enum Flags Flags;
```

A. Speicherbelegung eines *C*-Programmes

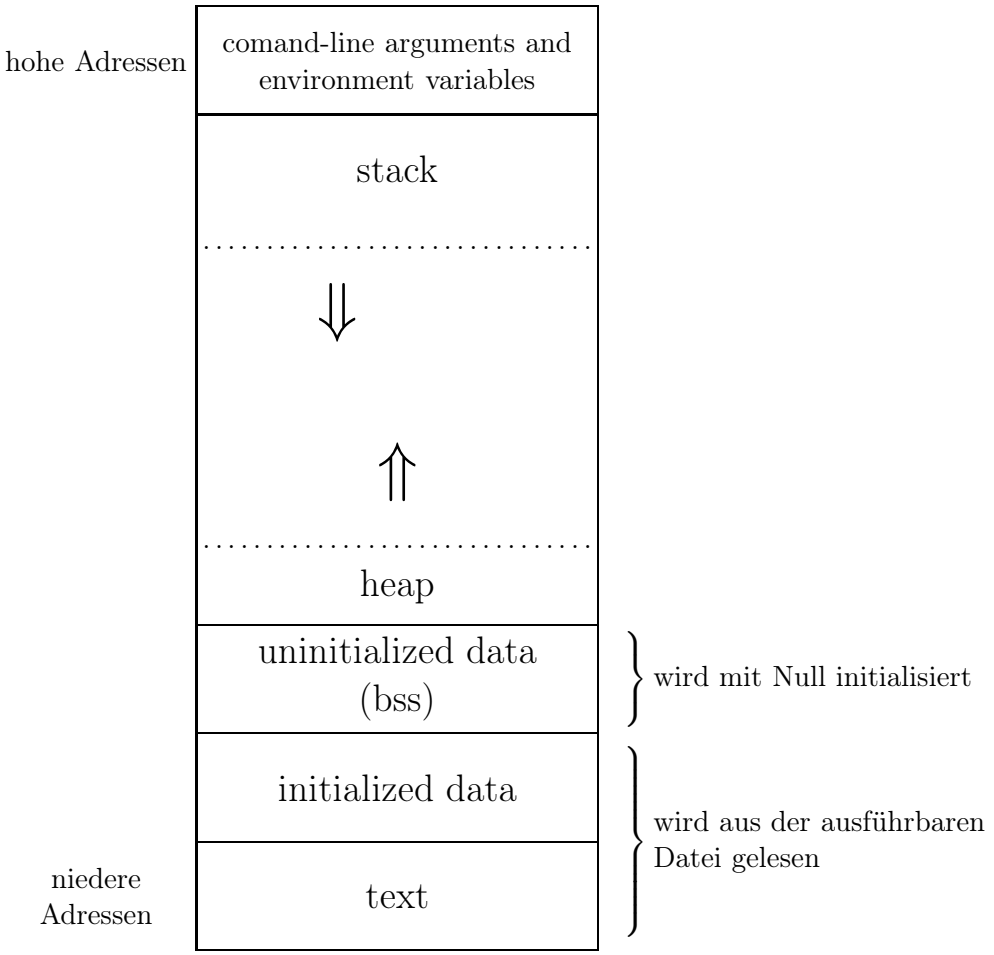


Abbildung A.1.: Typische Speicherbelegung eines *C*-Programmes

Abbildung A.1 zeigt die typische Speicherbelegung eines *C*-Programmes. Ein Speicherabbild ist aus folgenden Segmenten zusammengesetzt:

stack Variablen der Speicherklasse `automatic`, Rücksprungadressen von Funktionsaufrufen.

heap Dynamisch belegter Speicher (`malloc`, `calloc`, `free`).

uninitialized data Globale Variablen, die im Programm nicht initialisiert werden, zum Beispiel `int val[100]`. Diese Variablen werden beim Programmstart mit Null belegt.

initialized data Globale Variablen, die im Programm initialisiert wurden, zum Beispiel `int maxcount = 1000`.

text Maschinenbefehle des Programms.

Der Adreßbereich beginnt bei der virtuellen Adresse Null. Die Obergrenze ist abhängig von der Hardware-Plattform und dem Betriebssystem. Bei einem 32-Bit-Rechner unter ***Linux*** liegt die Obergrenze bei 3 GByte. Jeder Prozeß besitzt, unabhängig von seiner tatsächlichen Größe, einen derartigen virtuellen Adreßraum. In den meisten Fällen ist nur ein Bruchteil des Adreßraumes erforderlich und belegt. Diese tatsächlich belegten Teile werden durch das Betriebssystem mit Hilfe von Tabellen und Hardware-Unterstützung dem vorhandenen physikalischen Speicher zugeordnet.

Literaturverzeichnis

[C+97] Cannon u. a.: *Recommended C Style and Coding Standard.* Bell Labs, 1997. – Programmierrichtlinien für Fortgeschrittene.

[Han06] Hannover, RRZN (Hrsg.): *Die Programmiersprache C.* RRZN Hannover : RRZN-Klassifikationsschlüssel: SPR.C 1, 2006. – Nachschlagewerk mit Beispielen. Sehr gutes Preis-/Leistungsverhältnis.

[IH04] Isernhagen ; Helmke: *Softwaretechnik in C und C++.* 4. Aufl. Carl Hanser Verlag, 2004. – Sehr umfangreich. Geht deutlich über die Bedürfnisse der ersten beiden Semester hinaus. Sehr gut geeignet als anspruchsvolle Einführung und als Nachschlagewerk, auch in weiterführenden Veranstaltungen.

[KP97] Kelley ; Pohl: *A Book on C.* 4. Aufl. Addison Wesley Longman, 1997. – ISBN 0201183994. – Immer noch eine gute Einführungen in C.

[KP00] Kelley ; Pohl: *C by Dissection.* 4. Aufl. Addison Wesley Longman, 2000. – ISBN 0201713748. – Lehrbuch mit ausführlich erläuterten Beispielen.

Abbildungsverzeichnis

1.1. Programmentwicklungsschritte 1
1.2. Programmablaufplan „Umrechnungstabelle“ 15
1.3. `int`-Feld mit 5 Elementen 22

3.1. Interne Darstellung einer vorzeichenbehafteten `char`-Größe 39
3.2. Interne Darstellung einer vorzeichenbehafteten `int`-Größe (4 Byte) 40
3.3. Interne Darstellung einer `float`-Größe (4 Byte) 42
3.4. Interne Darstellung des Wertes 5,375 als `float`-Größe 42

4.1. `if-else`-Anweisung 51
4.2. `do-while`-Schleife 52
4.3. `switch`-Anweisung 53

5.1. Ablauf eines Funktionsaufrufes 60
5.2. Rekursive Funktionsaufrufe bei der Berechnung von 5! 71

6.1. Linearisierte Darstellung einer 8×8-Einheitsmatrix im eindimensionalen Speicher 74
6.2. Variable `a` und Zeiger `pa` auf diese Variable im Speicher 75

8.1. Interne Darstellung des ganzzahligen Wertes 1021 (32 Bits, oben) und des zugehörigen bitweisen Komplements (unten) 97

A.1. Typische Speicherbelegung eines *C*-Programmes 103

Tabellenverzeichnis

1.1. Programmentwicklungsschritte 2
1.2. Basisdatentypen in ***C*** 8
1.3. ***C***-Standardbibliothek, Header-Dateien (***C 11***) 10
1.4. Ersatzdarstellung einiger Steuerzeichen 13
1.5. Flags und Typangaben in `printf`-Formaten (Auswahl) 14
1.6. Klassifizierungsfunktionen für Einzelzeichen (Auswahl; `include`-Datei `ctype.h`) 24
1.7. Ein-/Ausgabefunktionen für Zeichenketten (Auswahl; `include`-Datei `stdio.h`) 24
1.8. Formatierte Dateieingabe und Dateiausgabe (`stdio.h`) 27
1.9. Modi zum Öffnen von Dateien 28

2.1. ***C***-Schlüsselwörter 34
2.2. Vorrang und Bindung der ***C***-Operatoren 37

3.1. Ganzzahlige Basisdatentypen (die Angabe `int` ist bei `short`, `long` und `long long` optional) 39
3.2. Reelle Datentypen 41

8.1. Bit-Operatoren, geordnet nach fallendem Vorrang 97
8.2. Wirkungsweise der Operationen `&`, `|` und `^` 98
8.3. Beispiele für die Anwendung von Bit-Operatoren 98
8.4. Beispiele für die Anwendung der Schiebeoperatoren 98

Programmlisten

1.1. Vollständiges *C*-Programm (`hallo.c`) mit einfacher Textausgabe 3
1.2. Ausgabe des Programmes `hallo.c` 3
1.3. *C*-Programm mit verschiedenen Kommentarformen 5
1.4. Einzelne Umrechnung von $°F$ nach $°C$ 7
1.5. Ausgabe des Programmes in Liste 1.4 7
1.6. Verwendung von `define`-Makros 11
1.7. Programm in Liste 1.6 nach dem Präprozessorlauf (ohne die Teile aus `math.h`) . 12
1.8. Umrechnungstabelle $°F$ nach $°C$ (Verwendung einer `while`-Schleife) 15
1.9. Umrechnungstabelle $°F$ nach $°C$ (Verwendung einer `for`-Schleife) 17
1.10. Umrechnungstabelle $°F$ nach $°C$ (Verwendung einer Funktion) 20
1.11. Verwendung eines Feldes 23
1.12. Klein- und Großbuchstaben abzählen 25
1.13. Formatierte Dateieingabe und Dateiausgabe 30
1.14. Eingabedatei `vektor.dat` 31
1.15. Ausgabedatei `vektor.aus` 31

4.1. Binärbaum durch Schachtelung von `if-else`-Anweisungen 51

5.1. Funktionsaufruf mit Wertübergabe 61
5.2. Verwendung eines Prototyps 62
5.3. Gebrauch von externen (globalen) Variablen 66
5.4. Modul 1: Vereinbarung globaler Variablen 67
5.5. Modul 2: Verwendung externer Variablen (aus Modul 1) 67
5.6. Modul 3: Vereinbarung einer globalen `static`-Variablen 68
5.7. Modul 4: Verwendung einer externen Variablen (aus Modul 3) 69
5.8. Rekursive Berechnung der Fakultät einer ganzen Zahl 70

6.1. Direkte und indirekte Ausgabe von Adressen und Werten 76
6.2. Verwendung von Zeigern zur Feldadressierung 79
6.3. Dynamische Speicherbelegung mit `calloc` 81
6.4. Kopieren und Wenden von Zeichenketten 83
6.5. Verwendung einer Funktion als Funktionsargument 85
6.6. Verwendung der Funktion `qsort` 87

7.1. Polarkoordinaten als Struktur 91
7.2. Adreßübergabe bei Strukturen 92
7.3. Darstellung von Vektoren der Länge 3 mittels Strukturen 94
7.4. Darstellung von Vektoren der Länge 3 mittels Strukturen (Fortsetzung) 95

8.1. Ausgabe der Bit-Darstellung eines `int`-Wertes (nach [KP97]) 99
8.2. Verwendung von Bit-Masken zur Handhabung von Benutzerrollen 100