# Programmieren in C

**Dipl.-Math. Michael Mair**
Dipl.-Math. Alexander Weiß

Sommersemester 2008

Numerische Mathematik für Höchstleistungsrechner
Institut für Angewandte Analysis und Numerische Simulation
Universität Stuttgart

# Inhaltsverzeichnis

# 0. Einführung

Die Programmiersprache ***C*** wurde in den 1970ern von K. Thompson und D.M. Ritchie für die Entwicklung von leicht portierbaren Betriebssystemen konzipiert. 1978 erschien das Buch „The C Programming Language“ von B.W. Kernighan und Ritchie; innerhalb weniger Jahre eroberte sich ***C*** eine große Fangemeinde für Programmierprojekte aller Art. Bis 1989 entstand aus dem klassischen K&R-***C*** ein ANSI-Standard für die Sprache, der heute von Compilern auf nahezu allen Betriebssystemen verstanden wird.

***C*** ist eine prozedurale Programmiersprache wie beispielsweise auch ***Fortran*** oder ***Pascal***. Im Gegensatz zum ursprünglichen ***Fortran*** wurde von vornherein ein dynamisches Programmiermodell unterstützt, im Gegensatz zu ***Pascal*** war ***C*** für hardwarenahe Programmierung ausgelegt.[1] Die Mächtigkeit von ***C*** als Allzweck-Programmiersprache gründet sich unter anderem auf die Verwendung von Zeigern – gleichzeitig ist das eine der häufigsten Fehlerquellen.

Aus ***C*** ist ***C++*** hervorgegangen, das ein objektorientiertes Programmierkonzept verfolgt, aber nicht auf die Sprachmittel von ***C*** verzichtet (einige davon aber überflüssig macht). Als rein objektorientierte Programmiersprache ohne viele der typischen Möglichkeiten zu „unsauberer“ Programmierung leitet sich auch ***Java*** aus ***C++*** und ***C*** ab. Umgekehrt sind einige bequeme und sinnvolle Erweiterungen (und Einschränkungen) aus ***C++*** 1989 in den ANSI-Standard und später im Rahmen eines ISO-Standards von 1999 wieder in ***C*** eingeflossen. ***C***99 wird aber noch nicht vollständig von allen Compilern „verstanden“. In diesem Jahr soll als ***C***05 ein neuer Standard erscheinen, der aber im Wesentlichen nur kleinere Korrekturen enthalten soll.

Warum ist es sinnvoll, ***C*** oder vergleichbare Programmiersprachen zu lernen? Es gibt für die meisten mathematischen Disziplinen Entwicklungsumgebungen, die wesentlich schneller zu sichtbaren Ergebnissen führen und „lästige“ Details vor dem Nutzer verbergen, z.B. im numerischen Umfeld ***Matlab***. Außerdem hat man in ***C*** wesentlich mehr Möglichkeiten unnötige Fehler zu machen…

Gründe für das Erlernen einer „richtigen“ Programmiersprache sind

- die Beschäftigung mit implementierungstechnischen Details: Man lernt, was eine Routine schneller, besser und stabiler macht.
- die Möglichkeit, zeitkritische Anwendungen zu beschleunigen.
- die Kleinigkeiten: Durch das Implementieren elementarer Funktionen aus verschiedenen Bereichen und darauf aufbauender Aufgaben wird man sich der vorhandenen Möglichkeiten bewusst und erlernt Techniken, die einem auch an anderer Stelle zugute kommen.

[1] Neuere ***Fortran***-Varianten unterstützen wesentlich mehr und gehen bis hin zu objektorientierten Ansätzen; von ***Pascal*** gab und gibt es Varianten, die durchaus zur „richtigen“ Softwareentwicklung taugten, und auch den Nachfolger Delphi, der objektorientiertes Programmieren ermöglicht.

Gründe für ***C*** sind die universelle Einsetzbarkeit, Programmgeschwindigkeiten nahe am Optimum, die gute Ausgangsbasis für den Umstieg zu ***C++***, ***Matlab***, ***Java*** sowie die Möglichkeit, von ***Matlab*** aus ***C***-Funktionen zu verwenden.

## Voraussetzungen

Im Wesentlichen braucht man für das Programmieren in ***C*** einen Editor, um die Programme zu erstellen, und eine Compiler-Umgebung. Diese enthält

- die ***C***-Standard-Bibliothek mit Funktionen für Aufgaben von Textausgabe, dem Umgang mit Zeit und Datum, Datei-Ein-/Ausgabe, mathematischen Funktionen bis hin zum Sortieren von Daten;
- den Präprozessor, der den Code für den Compiler aufbereitet, den eigentlichen Compiler, der ***C*** in ein rechnerabhängiges Assembler-Format übersetzt, den Assembler, der die Ausgabe des Compilers in Maschinensprache übersetzt und den Linker, der das entstandene Stück Code (evtl. mit anderen Modulen) zu einem ausführbaren Programm macht;
- in der Regel auch eine Dokumentation zur Standardbibliothek;
- oft einen Debugger zur Fehlersuche;
- manchmal einen Profiler zur Laufzeit-Optimierung.

Zusätzlich gibt es nützliche Hilfsprogramme, die die Entwicklung und Wartung von ***C***-Code vereinfachen oder die Fehlersuche erleichtern.

## Aufbau des Skripts

Eine Programmiersprache lernt man am besten, indem man programmiert. Daher sind in diesem Skript die einzelnen Abschnitte bei weitem nicht erschöpfend und es steht – im Rückblick – nicht immer alles da, wo es hingehört. Im Folgenden soll Stück für Stück jeweils so viel ***C*** vermittelt werden, wie zum Verständnis des jeweiligen Abschnitts notwendig ist. Wo Dinge vorgezogen sind, weil sonst die Beispiele im luftleeren Raum hängen würden, werden sie später an passender Stelle etwas ausführlicher erläutert. Der Text ist darauf ausgelegt, die grundlegenden Sprachmittel und Konzepte so schnell wie möglich zu vermitteln.

Neben kurzen Beispielen im Fließtext gibt es auch längere Beispielprogramme nach den eigentlichen Lektionen.

Die Verwendung der Compiler-Dokumentation sowie von Büchern oder evtl. auch anderen Skripten zum tieferen Verständnis sowie zur Lösung der Aufgaben kann sinnvoll sein.

Der eigentlichen Sprachbeschreibung folgt ein Kapitel zum Thema Datei-Operationen; anschließend werden nützliche Programmierwerkzeuge besprochen.

Zum Abschluss werden Grundzüge der Optimierung besprochen.

Im Anhang finden sich

- eine Einführung in die im CIP-Pool verwendete Shell *bash*; dort werden grundlegende Konzepte und einige Befehle erklärt. Wer sich noch nicht mit der Shell auskennt, sollte sich dieses Kapitel anhand der allerersten Übung erarbeiten.
- zwei Beispiele zur gemeinsamen Verwendung von ***C*** und ***Matlab***.
- ein Überblick über die Funktionenfamilien `printf` und `scanf` zur formatierten Ausgabe bzw. Eingabe. Nachdem ***C*** keine Sprachmittel für Aus- und Eingabe kennt, ist es sinnvoll, sich mit den entsprechenden Funktionen zu beschäftigen. Meistenteils dürften die beiden Tabellen in der Mitte des Kapitels genügen, aber für spezielle Zwecke oder zur Fehlersuche kann es notwendig sein, die formale Beschreibung durchzuarbeiten.

Danach folgt ein kurzes Literaturverzeichnis.

**Eine große Bitte zum Schluss:** Wer auf einen Fehler oder eine Ungereimtheit stößt, egal ob sachlich oder sprachlich, oder wer auf einen Begriff stößt, der ihm/ihr nicht klar ist: Bitte gebt mir Bescheid[2], damit ich die entsprechende Stelle verbessern kann!

## Anmerkungen zur Notation

***C***-Quellcode und Namen von Befehlen, Funktionen, Variablen sowie kurze Codeschnipsel werden in `Schreibmaschinenschrift` gedruckt. Auslassungen werden hierbei mit vier Punkten markiert, `....`, weil drei Punkte in ***C*** eine spezielle Bedeutung haben.

So sehen *Fachbegriffe* aus, so *Werkzeuge*.

Die Beispielprogramme sind grau hinterlegt und stehen zum Herunterladen bereit.

[2] Entweder über den Kursleiter oder direkt unter `Michael.Mair@Lustige-Mathematiker.de`

# Teil I.

# Die Sprache

## Übersicht

*Die Programmiersprache **C** besteht aus zwei Teilen: der eigentlichen, relativ schlanken Sprache und der Standardbibliothek. Ohne Bibliotheksfunktionen hat **C** nicht einmal die Möglichkeit zur portablen Textein- und -ausgabe.*

*Wir werden uns in den folgenden Abschnitten zunächst mit der Sprache beschäftigen und fangen beim allgemeinen Programmaufbau und Funktionen an. Danach geht es weiter beim Kontrollfluss, also Bedingungen und Iterationsvorschriften. Bereits an dieser Stelle kann man (abgesehen von Ein- und Ausgabe) alle Programme schreiben – aber noch nicht bequem. Nach einem Ausflug zu den Aufgaben von Präprozessor und Compiler sehen wir uns die in **C** zur Verfügung stehenden Datentypen und die darauf anwendbaren Operationen an. Abschließend betrachten wir Funktionen, die wir von Anfang an verwenden werden, noch einmal genauer.*

*Einige der Standardbibliotheks-Funktionen werden uns im Laufe dieses Teils begegnen, weitere finden sich im nächsten Teil bzw. im Anhang.*

# 1. Programmaufbau und Funktionen

## 1.1. Allgemeiner Aufbau

Die Anweisungen eines Programms werden in einer Datei dateiname.c geschrieben. Das Programm hat dabei folgenden Aufbau:

```
Include-Anweisungen
globale Deklarationen
int main(...)
{
    lokale Deklarationen
    Anweisungen
}

Typ function1(...)
{
    lokale Deklarationen
    Anweisungen
}

Typ function2(...)
{
    lokale Deklarationen
    Anweisungen
}
...
```

**Erklärungen:**

- int main(...): Das Hauptprogramm. Es wird als erstes ausgeführt, wenn das Programm gestartet wird.
- function1(...),function2(...),... : Das sind optionale Unterprogramme, die z.B. in main ausgeführt werden.
- Jedes ***C***-Programm muss genau ein Hauptprogramm mit Namen „main“ enthalten. Die Namen der Unterprogramme sind frei wählbar.

## 1.2. Erstes *C*-Programm

Das klassische erste ***C***-Programm sieht folgendermaßen aus:

```
/*********************************************************
**                                                      **
**  helloworld - mein erstes C-Programm                 **
**                                                      **
*********************************************************/

/* Include-Datei fuer Standard-IO-Funktionen (printf) */
#include <stdio.h>

/** Hauptprogramm **/
int main (void)
{
    printf("Hello, world\n");

    return 0;
}
```

**Erklärungen:**

- Alles zwischen den Zeichen `/*` und `*/` wird als Kommentar betrachtet und zählt nicht zum eigentlichen Programm.
- ***C*** verfügt als Sprache nicht über Ausgabebefehle; deshalb müssen mit der Präprozessoranweisung `#include <stdio.h>` die Standard-Eingabe- und Ausgabe-Funktionen bekannt gemacht werden, zu denen auch die weiter unten verwendete Funktion `printf()` gehört.
- Dann kommen wir zum eigentlichen Programm: Die Funktion `main()` erhält keine Eingabeparameter – deshalb das `void` in der Argumentliste, die zwischen den runden Klammern erscheint – und liefert einen Ganzzahl-Wert (hier 0) zurück – deshalb das `int` vor dem Funktionsnamen.[1]
- Nach diesen Informationen zur Funktion `main()`kommt dann zwischen den geschweiften Klammern der Funktionsrumpf, in dem die eigentliche Arbeit getan wird. Die geschweiften Klammern markieren in ***C*** immer einen Anweisungsblock.

[1] `int` ist der Standard-Ganzzahl-Datentyp in ***C***. `main` hat deshalb keine leere Rückgabe, weil z.B. beim Aufruf des Programmes aus einer Shell ein Rückgabewert erwartet wird, der Aufschluss über Erfolg (0) bzw. Misserfolg (andere Werte) geben soll. Es ist nicht vom Standard vorgesehen, aber für manche Compiler auch möglich, `main` als vom Typ `void main()` zu verwenden; dann braucht man keine `return`-Anweisung. Das führt meistens zu einer Warnung und wird oft vom Compiler intern „korrigiert“. „`void main()`“ findet sich oft noch in älteren Texten zu ***C***, sollte aber *nicht* verwendet werden, weil es schlicht falsch ist.
`main` kann auch eine andere Argumentliste haben, die es ermöglicht, beim Programmaufruf Parameter an das Programm zu übergeben, siehe Abschnitt 6.2.

- Funktionen stehen immer in Anweisungsblöcken; innerhalb von Funktionen darf man so viele Blöcke abteilen, wie man möchte. Die Funktion `main()` tut nicht besonders viel: Sie ruft ihrerseits die Funktion `printf()` auf, der als Argument eine Zeichenkette (*String*) übergeben wird.

- In dem übergebenen String befindet sich noch eine unerklärte Zeichenfolge: Durch den Backslash `\` wird angegeben, dass ein Steuerzeichen gewünscht wird; `\n` besagt, dass nach der Ausgabe `Hello, world` eine neue Zeile begonnen werden soll[2].Mehr zu `printf` findet sich in den folgenden Kapiteln sowie im Anhang C.2.

- Der Aufruf der Funktion wird – wie jede Anweisung in ***C***– durch ein Semikolon abgeschlossen.

- Die Anweisung `return` beendet die Ausführung der Funktion und liefert einen Wert von dem Typ zurück, der als Rückgabe-Datentyp angegeben wurde, also hier den `int`-Wert 0[3].

***C***-Quellcode-Dateien haben in der Regel einen Namen, der auf „.c“ endet, z.B. „helloworld.c“ für obiges Programm. Der für den Kurs verwendete Compiler *gcc* (siehe 3.2) kann auch für andere Programmiersprachen verwendet werden. Die Endung „.C“ beispielsweise führt deshalb dazu, dass die Datei als ***C++***-Quellcode interpretiert und intern an den ***C++***-Compiler *g++* weitergegeben wird. Eine ***C***-Quellcode-Datei, die vom Compiler kompiliert werden kann, nennt man *Übersetzungseinheit*.

In ***C***-Quellcode spielen Leerzeichen, Tabstopps und Zeilenumbrüche zwischen den Schlüsselwörtern, Klammern, Operatoren usw. keine besondere Rolle; wir könnten das Programm (ohne Kommentare) auch so schreiben:

```
#include <stdio.h>
int
 main
  (void) { printf("Hello, world\n"
); return 0; }
```

Nur die `#include`-Anweisung muss auf einer eigenen Zeile stehen; die Zeichenkette darf keinen Zeilenumbruch enthalten und reagiert logischerweise auf eingefügte Leerzeichen.

Funktionsnamen müssen innerhalb ihres Gültigkeitsbereiches, in der Regel im ganzen Programm oder innerhalb einer Datei, eindeutig sein. Der Funktionsname `main` ist hierbei für das Hauptprogramm reserviert, d.h. innerhalb eines Programmes *muss* es genau eine Funktion `main()` geben, da mit dieser Funktion immer angefangen wird.

---

[2] Die letzte Ausgabe eines Programmes sollte immer durch `\n` abgeschlossen werden, um sicherzustellen, dass sie auch tatsächlich „ankommt“.

[3] Gibt `main` einen `void`-Wert zurück, so lautet die Anweisungszeile `return;` und kann auch weggelassen werden; die schließende geschweifte Klammer wird dann als Funktionsende interpretiert. Für alle anderen Rückgabe-Datentypen muss dagegen immer die Funktion durch `return` plus Wert vom entsprechenden Typ beendet werden.
***C***99 erlaubt für `main` (und nur für `main`) das Weglassen der `return`-Anweisung.

## 1.3. Unterprogramme

Eine Quellcode-Datei wird von oben nach unten gelesen, deswegen müssen Funktionen (und Variablen) bekannt sein, bevor das erste Mal auf sie zugegriffen wird:

```
/*********************************************************
**                                                      **
**  SquareNumber1 - Quadriere eine Zahl                 **
**                                                      **
*********************************************************/

/* Include-Datei fuer Standard-IO-Funktionen (printf) */
#include <stdio.h>

/* Funktionsprototyp: Bekanntmachen von SquareNumber */
double SquareNumber (double number);


/** Hauptprogramm **/
int main (void)
{
    double zahl, quadratzahl;

    zahl = 13.57;
    quadratzahl = SquareNumber(zahl);

    printf("%f * %f = %f\n",zahl,zahl,quadratzahl);

    return 0;
}

/*********************************************************
** SquareNumber - liefere das Quadrat einer Zahl
**
** Argumente: Fliesskommazahl number
**
** Rueckgabe: Fliesskommazahl (number*number)
*/
double SquareNumber (double number)
{
    double result;

    result = number*number;

    return result;
}
```

**Erklärungen:**

- Die Typbezeichnung `double` bezeichnet doppelt genaue Fließkommazahlen.
- Die Funktion `SquareNumber()` nimmt eine Fließkommazahl als Argument und liefert mit der `return`-Anweisung ihr Quadrat – ebenfalls als Fließkommazahl – zurück. Die Funktion wird vor `main` (global) deklariert, der Funktionscode wird aber erst nach `main` aufgeschrieben.
- In ähnlicher Weise werden am Anfang von `main` die beiden Fließkomma-Variablen `zahl` und `quadratzahl` (lokal) deklariert, aber erst weiter unten wird ihnen ein Wert zugewiesen, wobei in `quadratzahl` der Rückgabewert der Funktion `SquareNumber` steht.
- Das Zeichen `%` in dem String, der `printf` übergeben wurde, zeigt an, dass ein formatierter Wert – in diesem Fall mit `%f` eine Fließkommazahl – ausgegeben werden soll; dieser Wert muss dann `printf` als weiteres Argument übergeben werden. Hierbei müssen mehrere Werte in der Reihenfolge nach dem String stehen, in der sie ausgegeben werden sollen.
- Unterprogramm wird definiert (dieses hat am Schluss kein Semikolon (;), dafür aber einen Anweisungsblock). Der Übergabeparameter muss nicht deklariert werden.

Bemerkung:

```
scanf("%d") liest int ein.
```

Alternativ können Deklaration (Bekanntmachung) und Definition von Funktionen und Variablen gleich am Anfang ausgeführt werden:

```
/*********************************************************
**                                                      **
**  SquareNumber2 - Quadriere eine Zahl                 **
**                                                      **
*********************************************************/

/* Include-Datei fuer Standard-IO-Funktionen (printf) */
#include <stdio.h>

/*********************************************************
** SquareNumber - liefere das Quadrat einer Zahl
**
** Argumente: Fliesskommazahl number
**
** Rueckgabe: Fliesskommazahl (number*number)
*/
double SquareNumber (double number)
{
    return number*number;
}
```

```
/** Hauptprogramm **/
int main (void)
{
    double zahl = 13.57;
    double quadratzahl = SquareNumber(zahl);

    printf("%f * %f = %f\n",zahl,zahl,quadratzahl);

    return 0;
}
```

Insbesondere die gleichzeitige Deklaration und Wertzuweisung per Funktion ist aber etwas unübersichtlich. Im obigen Beispiel wird mit den Variablen `zahl` und `quadratzahl` nicht viel getan, deshalb könnte `main` auch so

```
int main (void)
{
    double zahl = 13.57;

    printf("%f * %f = %f\n",zahl,zahl,
                            SquareNumber(zahl));
}
```

oder gar so

```
int main (void)
{
    printf("%f * %f = %f\n",13.57,13.57,
                            SquareNumber(13.57));
}
```

aussehen.

Noch ein Beispiel:

```
/* Abs-Betrag - liefert den Absolutbetrag einer double-Zahl */
double AbsBetrag(double number)
{
    if(number >= 0.0)
    {
        return (number);
    }
    else
    {
        return (-number);
    }
}
```

**Erklärungen:**

- if überprüft eine Bedingung (in Klammern). Ist die Bedingung wahr, wird der folgende Block ausgeführt, sonst der Block nach else.
- mögliche Vergleiche: <, > , <=, >=, ==, !=
- „0.0“ ist die Zahl 0 als double. „0“ wäre die Zahl 0 als Int
- An Unterprogramme und return können nicht nur Variablen zurückgegeben werden, sondern auch Ergebnisse von Operationen und Funktionen.

## 1.4. Kommentare, Auslassungen, mögliche Fehlerquellen

> Real programmers don’t write comments – it was hard to write, so it should be hard to understand.

Die Gestalt der ***C***-Kommentare ist aus den vorherigen Programmbeispielen bekannt: `/*` *Kommentartext* `*/`. Sie dürfen nicht geschachtelt werden, was Anlass zu Fehlern geben kann; wenn man, um etwas auszuprobieren, eine Programmpassage auskommentiert, die bereits einen Kommentar enthält, kann der Kommentar früher zu Ende sein als geplant. Umgekehrt kann es vorkommen, dass man einen Kommentar beginnt, aber zu beenden vergisst – dann beendet erst der „nächste“ Kommentar den aktuellen. Solche Fehler können durch die Verwendung von Editoren vermieden werden, die Syntax-Highlighting beherrschen und beispielsweise Kommentare im Gegensatz zum anderen Programmtext kursiv darstellen.

Mit ***C***99 können auch die sogenannten ***C***++-Kommentare verwendet werden; diese sind einzeilig und werden durch `//` begonnen. Sie eignen sich besser für kurze Anmerkungen am Ende von Zeilen oder einzeilige Kommentare. Im Allgemeinen sollte man die ***C***-Kommentare auch weiterhin für mehrzeilige Kommentare verwenden. Beim Mischen muss man vorsichtig sein, um nicht seltsame Ergebnisse hervorbringen zu können. Hier ein Beispiel für die Gefahren bei der Verwendung von ***C***++-Kommentaren in Verbindung mit ***C***-Kommentaren:

```
/**** Include-Dateien ****/

#include <stdio.h>  // fuer printf
#include <math.h>   // fuer Wurzelfunktion sqrt

void funktion_mit_ausgabe()
{
    printf("Nicht auskommentiert\n"); /* Mehrzeiliger
                                       ** Kommentar ohne
                                       ** Anfang! */
    printf("Das hier auch nicht\n");  /* _Schlechter_
                             /* Kommentarstil; es klappt
                                /* zufaelligerweise. */
    printf("Dito\n");    /* Narrensicherer K-Stil    */
```

```
                            /* funktioniert immer; viel */
                            /* zu muehsam gegenueber // */
}

void funktion_ohne_ausgabe()
{
//  printf("Nicht auskommentiert\n"); /* Mehrzeiliger
                                       ** Kommentar ohne
                                       ** Anfang! */
//  printf("Das hier auch nicht\n");  /* _Schlechter_
                               /* Kommentarstil; es klappt
                                  /* zufaelligerweise. */
//  printf("Dito\n");    /* Narrensicherer K-Stil    */
                            /* funktioniert immer; viel */
                            /* zu muehsam gegenueber // */
}
```

Welchen Kommentarstil man bevorzugt, ist Geschmackssache - wichtig ist, dass man kommentiert und dass man seine Kommentare auch immer aktualisiert! Nichtsdestotrotz sollte darauf geachtet werden, dass ein Kommentarstil nicht allzu aufwendig in der Pflege ist. Die Art, wie die Programme helloworld und squarenumber am Datei-Anfang kommentiert wurden, ist sehr aufwendig, wenn es hinterher immer noch hübsch aussehen soll; jedes Neu-Umbrechen einer Zeile verlangt ein wenig Arbeit, längere Kommentare, die auch die Programm-Geschichte enthalten, kosten verhältnismäßig viel Zeit beim Erstellen.

Im Prototyp von `funktion_mit_ausgabe()` und `funktion_ohne_ausgabe()` wurde auf die Angabe `void` für die Parameter verzichtet – das ist durchaus zulässig. Auf die Angabe des Rückgabetyps sollte dagegen nicht verzichtet werden. ANSI-***C*** hat wie auch K&R-***C*** die Regelung, dass alles, was nicht deklariert ist, von dem ganzzahligen (=*Integer*-)Typ `int` ist; fehlende Integer-Rückgaben werden nicht notwendigerweise angemahnt – in ***C***99 dagegen muss der Typ von Variablen und der Rückgabetyp von Funktionen immer angegeben werden und die Rückgabe sollte immer entsprechend erfolgen. Für `void`-Funktionen wäre die entsprechende Rückgabe leer und würde deshalb durch `return;` erfolgen. Das Erreichen der schließenden Klammer `}` eines Funktionsblocks wird als `return;` interpretiert. Eine mögliche Fehlerquelle ist bei Funktionen mit nichtleerem Rückgabetyp, die richtige Rückgabe zu vergessen. Abhilfe schafft das obligatorische Setzen eines Rückgabewertes in der Zeile vor dem Funktionsende. Dadurch wird vermieden, dass ein zufälliger bzw. Standard-Wert zurückgegeben wird, der fehlerfreie Funktion vorspiegelt.

Es gibt zwar keine Beschränkung der Zeilenlänge, aber wenn eine Zeile „zu lang“ ist, kann man wie oben erwähnt ohne Probleme eine neue Zeile anfangen und dort die Anweisung fortsetzen. Eine Ausnahme bilden Strings, da bei ihnen ja der Zeilenumbruch mit in die String-Definition aufgenommen würde. Das Zeilenende kann vor dem Compiler verborgen werden, indem man als letztes Zeichen in der Zeile einen Backslash setzt:

```
"Das ist ein\
gueltiger String"
```

Nach dem Backslash darf kein anderes Zeichen mehr in der Zeile stehen; speziell Leerzeichen oder Tabulatoren sind hier „gefährlich“. Im Fall von Strings gibt es eine weitere, sicherere Möglichkeit,

nämlich den String in dieser Zeile zu beenden und in der nächsten gleich wieder zu beginnen:

```
"Das ist ein"
"gueltiger String"
```

Nachdem der String am Zeilenanfang bzw. beim neuen öffnenden Anführungszeichen gleich weitergeht, werden führende Leerzeichen und Tabstopps auch mit berücksichtigt – in obigem Fall fehlt ein Leerzeichen: Der Compiler liest beide Male "Das  ist    eingueltiger    String"…

## 1.5. Gültigkeitsbereich von Variablen

Variablen und Funktionen haben einen Gültigkeitsbereich (*scope*), innerhalb dessen sie „leben“. Hierbei unterscheidet man zwischen dem globalen Gültigkeitsbereich, dem Datei-Gültigkeitsbereich, dem Funktions-Gültigkeitsbereich und dem Block-Gültigkeitsbereich. Funktionen können nur global oder auf Dateiebene gültig bzw. bekannt sein; sie können nicht innerhalb anderer Funktionen definiert werden. Falls gleiche Namen vorkommen, gilt, dass die lokale Variante gültig ist, solange sie lebt, und die im größeren Gültigkeitsbereich lebende für die Dauer ihrer Lebenszeit überschattet (*shadowing*).

```
int dateiweitbekannt = 1; /* Ab hier lebt dateiweitbekannt,
                           ** innerhalb der ganzen Datei */
int i = 0; /* Ab hier lebt i, innerhalb der ganzen Datei */

int funktion (double argument1, int argument2)
{   /* Ab hier kann funktion() sich selbst aufrufen,
    ** ab hier leben argument1 und argument 2 */
    int i; /* Ab hier lebt i, innerhalb von funktion */

    i = dateiweitbekannt; /* i=1 */

    {
        int i = argument2; /* Ab hier lebt i,
                            ** innerhalb des Blocks. */
        /* i=argument2 */
    }
    /* i=1 */
}
/* i=0;
** funktion() kann jetzt von aussen aufgerufen werden;
** gaebe es weiter oben einen Prototyp von funktion(), koennte
** funktion() unterhalb des Prototyps aufgerufen werden. */
```

Es ist sinnvoll, die Variablen erst in dem Gültigkeitsbereich einzuführen, in dem sie gebraucht werden und so wenige Variablen wie möglich mit dateiweiter bzw. globaler Gültigkeit zu versehen.

| | | | |
|---|---|---|---|
| **auto** | **enum** | **restrict** | **unsigned** |
| **break** | **extern** | **return** | **void** |
| **case** | **float** | **short** | **volatile** |
| **char** | **for** | **signed** | **while** |
| **const** | **goto** | **sizeof** | **_Bool** |
| **continue** | **if** | **static** | **_Complex** |
| **default** | **inline** | **struct** | **_Imaginary** |
| **do** | **int** | **switch** | |
| **double** | **long** | **typedef** | |
| **else** | **register** | **union** | |

**Tabelle 1.1.:** ***C***-Schlüsselworte. Neben den Schlüsselworten empfiehlt es sich auch, Makros und Typdefinitionen der Standard-Bibliothek zu vermeiden. Außerdem definieren `<stdbool.h>` und `<complex.h>` die Pseudo-Schlüsselworte `bool`, `true` und `false` bzw. `complex` und `imaginary`.

## 1.6. Variablen- und Funktionsnamen

In ANSI-***C*** müssen die in einem Block (also auch innerhalb einer Funktion) verwendeten Variablen am Anfang des Blockes (bzw. der Funktion) deklariert (bekanntgemacht) werden, in ***C***99 kann das an beliebiger Stelle vor bzw. bei ihrem ersten Einsatz geschehen. Das führt, wie wir später sehen werden, zu einer bequemeren Handhabung von Variablen, die man erst an einer bestimmten Stelle benötigt, aber auch zu der Gefahr, dass man ein wenig die Übersicht verliert, welche Variablen verwendet werden.

Variablen- und Funktionennamen dürfen in ***C*** aus Ziffern und Buchstaben bestehen und müssen mit Buchstaben anfangen. Die in ***C*** zulässigen Buchstaben sind

```
abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ_
```

wobei der Unterstrich _ den Buchstaben zugerechnet wird. Die zulässigen Ziffern sind

```
0123456789
```

In ***C*** wird zwischen Groß- und Kleinschreibung unterschieden. Wie schon erwähnt, dürfen Funktionsnamen in ihrem Gültigkeitsbereich nicht mehrfach vergeben werden. Das gleiche gilt für Variablennamen, wobei es hier mehr Möglichkeiten für den Gültigkeitsbereich gibt. Die Namen von Funktionen und Variablen werden gemeinsam betrachtet, d.h. es kann nicht gleichzeitig eine Funktion und eine Variable gleichen Namens geben, so dass man auch auf beide zugreifen kann! Außerdem dürfen als Namen nicht ***C***-Schlüsselworte zugerechnet werden. Die vom ***C***-Standard festgelegten Schlüsselworte finden sich in Tabelle 1.1. Auf „Namenskonventionen“ wird im Abschnitt 3.1.1 eingegangen.

***Programmlisting 1.1:*** volumina.c

```
/*****************************************************************
**
** volumina -- Berechne Kreis- und Kugelvolumen
**
** Geschichte:
**
**             2004-04-16: Kleinere Aenderungen durch M. Mair
**
**             2004-04-15: Neues Leben eingehaucht durch J. Werner
**
**             Finstere Vergangenheit: Irgendwo abgestaubt
**
*****************************************************************/


/*** INCLUDES & DEFINES ******************************************/

#include <stdio.h>  // printf() und scanf()


/*** Funktionsprototypen *****************************************/

double BerechneFlaeche (double); /* bei Prototypen reicht bei den   */
double BerechneVolumen (double); /* Argumenten die Angabe der
                                 ** Variablentypen. Angabe der Namen
                                 ** tut aber nicht weh; man kann bei
                                 ** mehreren Argumenten schneller
                                 ** herausfinden, welches nun wofuer
                                 ** gut ist.
                                 */



/*****************************************************************
**
** Hauptprogramm
**
** Argumente: Keine
**
** Rueckgabe: 0 .. kein Fehler
**
*****************************************************************/

int main (void)
{
    double radius, flaeche, volumen;

    printf("Beispielprogramm zur Berechnung von Kreisflaechen und");
    printf(" Kugelvolumina\nBitte geben Sie den Radius an: ");

    scanf("%lf", &radius); // Eingabe einlesen, Wert -> radius
```

```
    printf("Berechne die Flaeche eines Kreises mit dem Radius");
    printf(" r = %f mm\n", radius);

    flaeche = BerechneFlaeche(radius);
    printf("Die Flaeche A betraegt %f mm2\n", flaeche);

    volumen = BerechneVolumen(radius);
    printf("Das Volumen V betraegt %f mm3\n", volumen);

    return (0); // 0: Programm erfolgreich abgeschlossen
}

/*****************************************************************
**
** BerechneX - berechnet das n-dimensionale Kugelvolumen
**
** BerechneFlaeche: n=2
** BerechneVolumen: n=3
**
** Potenzielle Fehlerquelle: pi nicht genau genug.
** Abhilfe: Konstante PI aus <math.h> verwenden.
**
** Argumente: r .. Radius der Kugel in doppelter Genauigkeit
**
** Rueckgabe:   .. Volumen der Kugel in doppelter Genauigkeit
**
*/

double BerechneFlaeche (double r)
{
    double pi = 3.141592654; // Ginge genauer

    return(pi * r * r);
}

double BerechneVolumen (double r)
{
    double pi = 3.141592654; // Ginge genauer

    return((4.0 / 3.0) * pi * r * r * r);
}
```

# 2. Kontrollfluss

## 2.1. Bedingungen

Bis jetzt waren wir recht eingeschränkt auf fest vorgegebene Variablen sowie einen linearen Programmverlauf. Das interaktive Einlesen von Variablen ist mit der Funktion `scanf()` aus der Standard-Bibliothek möglich; unterschiedliche Entscheidungen können mit Hilfe der `if` -Anweisung getroffen werden. Die Syntax für die Verwendung von `if` ist *if Bedingung Anweisung;*. In der Regel steht hierbei die Bedingung in Klammern. Statt einer Anweisung kann man auch einen Anweisungsblock angeben, *if (Bedingung) {Anweisungsblock}*, unter Umständen gefolgt von *else AlternativeAnweisung;* bzw. *else {AlternativerAnweisungsblock}*.

```
/*********************************************************
**                                                      **
**  SquareNumber3 - Quadriere eine eingegebene Zahl     **
**                  und vergleiche sie mit ihrem Betrag **
**                                                      **
*********************************************************/

/* Include-Dateien */
#include <stdio.h>                      /* printf, scanf */

/*********************************************************
** SquareNumber - liefere das Quadrat einer Zahl
**
** Argumente: Fliesskommazahl number
**
** Rueckgabe: Fliesskommazahl (number*number)
*/
double SquareNumber (double number)
{
    return number*number;
}


/** Hauptprogramm **/
int main (void)
{
    double zahl, quadratzahl;
```

```
    /* Zahl einlesen... */
    printf("Geben Sie eine Fliesskommazahl ein:  ");
    scanf("%lf",&zahl);

    /* ... quadrieren... */
    quadratzahl = SquareNumber(zahl);

    /* ... auswerten... */
    if (quadratzahl < 1.0) {
        double test;

        printf("Das Quadrat von %f ist betrags"
               "maessig kleiner als %f selbst.\n",
               zahl, zahl);

        test = zahl - quadratzahl;
        if (test < 0)
            printf(" %f ist kleiner 0\n", zahl);
    }
    else { /* quadratzahl >= 1.0 */
        if (quadratzahl == 1.0)
            printf("Betrueger!\n");
        else /* quadratzahl > 1.0 */
            printf("Wie langweilig...\n");
    }

    /* ...ausgeben. */
    printf("%f * %f = %f\n", zahl, zahl, quadratzahl);

    return;
}
```

Zu beachten ist bei `scanf`, dass als Argument nicht eine Variable direkt, sondern indirekt über ihre Adresse übergeben wird (das Berechnen der Adresse wird bewirkt durch die Anwendung des Operators & auf die Variable); die Gründe hierfür werden später in Abschnitt 4.7 bzw. kurz im Anhang C.2 erläutert. Der übergebene Formatstring wird genau so eingelesen wie angegeben, d.h. wenn man für die Frage nach einer Fließkommazahl als Formatstring `Hallo Du %lf` angibt, muss für das Einlesen der Zahl 1.5 tatsächlich `Hallo Du 1.5` eingegeben werden. Wenn man außer den Formatangaben noch andere Zeichen in den String schreibt, erwartet `scanf` eine passende Eingabe. Bei der Forderung nach einem Zeilenvorschub `\n` muss der Eingabepuffer vor dem nächsten Aufruf von `scanf` geleert werden. Wie das funktioniert, findet sich im Anhang zu `scanf`, C.3.

Die Anweisung bzw. der Anweisungsblock nach der `if`-Abfrage wird ausgeführt, wenn die Bedingung, die abgefragt wird, erfüllt ist. ***C*** kennt ursprünglich keinen Typ für Wahrheitswerte (*boolean*), deshalb liefert der Vergleich den Wert 1 für wahr und den Wert 0 für falsch. Die Anweisung(en) wird/werden ausgeführt, wenn die Auswertung des Ausdrucks nach `if` einen Wert un-

gleich 0 ergibt. Falls sich 0/falsch ergibt und eine `else`-Anweisung folgt, wird der Block/die Anweisungszeile nach `else` ausgeführt.

`if`- bzw. `if .... else`-Anweisungen bieten einige Fallstricke: Zum einen ist es sehr leicht, versehentlich statt des Vergleichsoperators `==` den Zuweisungsoperator `=` zu verwenden, was dazu führen kann, dass die Bedingung immer als wahr oder immer als falsch interpretiert wird,

```
if (i==0) { /* wird ausgefuehrt, wenn der Wert der Klammer
             ** ungleich 0 ist, d.h. wenn der Vergleich
             ** wahr liefert, also wenn i gleich 0 ist */ }
if (i=0)  { /* wird ausgefuehrt, wenn der Wert der Klammer
             ** ungleich 0 ist, d.h. wenn das Ergebnis der
             ** Zuweisung (das gleich i ist) ungleich 0
             ** ist, also nie. */ }
if (i=1)  { /* wird ausgefuehrt, wenn der Wert der Klammer
             ** ungleich 0 ist, d.h. wenn das Ergebnis der
             ** Zuweisung (das gleich i ist) ungleich 0
             ** ist, also immer. */ }
```

und zum anderen kann die Optik des Programms einen falschen Eindruck von der Arbeitsweise vermitteln:

```
if (langervariablenname==zutestenderwert)
    tuwas;  // wird in der if-Anweisung ausgefuehrt
    tunochwas; // wird immer ausgefuehrt
tunochwasanderes; // wird immer ausgefuehrt
if (langervariablenname==zutestenderwert); // leere Anwsg.
{ // nach der leeren Anweisung ist if () ; zuende
    tuwas;  // wird immer ausgefuehrt
    tunochwas; // wird immer ausgefuehrt
} // lediglich ein Anweisungsblock, gehoert nicht zu if
tunochwasanderes; // wird immer ausgefuehrt
if (bedingung1)
    if (bedingung2) {
        tuwas;  // bedingung1 und bedingung2
        tunochwas; // dito
    }
else // dieses else gehoert zu bedingung2!!!
    tunochwasanderes; // bedingung1, aber nicht bedingung2
```

In ***C***99 gibt es für Wahrheitswerte den Datentyp `_Bool`, der nur die Werte 0 und 1 annehmen kann; die seltsame Schreibweise rührt daher, dass in alten ***C***-Programmen eigene BOOLEsche Datentypen definiert wurden. Wenn man neue Programme entwickelt, stehen beim Einbinden von `<stdbool.h>` die Definitionen `bool`, `true` und `false` zur Verfügung.

## 2.2. Mehrfachanweisungen

Der Standard-Ganzzahl-Datentyp in ***C*** ist `int` (wie *Integer*), der vorzeichenbehaftete ganze Zahlen in einem vom Rechnertyp abhängigen Zahlenbereich speichert, z.B. für ein 32-Bit-Integer im Bereich von $-2^{31}$ bis $2^{31}-1$.[1] Wenn wir für ein solches Integer mehrere Möglichkeiten haben, sieht eine Mehrfachanweisung mit `if` und `else` folgendermaßen aus:

```
int index;

....


if (index==0) {
    aktion0;
} else if (index==1) {
    aktion1;
} else if (index==2) {
    ....
} else {
    standardaktion;
}
```

Für Abfragen dieser Art bietet ***C*** die `switch`-Anweisung an, die manchmal auch zu kürzeren Programmlaufzeiten führt:

```
int index;

....


switch (index) {
    case 0:
        aktion0;
        break;
    case 1:
        aktion1;
        break;
    case 2:
        ....
    default:
        standardaktion;
}
```

Die `break`-Anweisung führt zum Verlassen des `switch`-Blocks; vergisst man sie bzw. lässt sie aus, wird der Programmcode der Reihe nach ausgeführt, d.h. bei folgender Konstellation

```
switch (index) {
```

[1] Es stehen aber immer mindestens die Zahlen von $-(2^{15}-1)$ bis $-(2^{15}-1)$ ($=\pm 32767$) zur Verfügung.

```
        case 0:
            aktion0;
        case 1:
        case 2:
            aktion1;
        case 3: case 4: case 7: case 9:
            aktion2;
        default:
            standardaktion;
    }
```

wird in einer Art stufenweiser Initialisierung in jedem Fall `standardaktion` ausgeführt, für die `index`-Werte 0 bis 4, 7 und 9 zusätzlich vorher `aktion2`, für 0 bis 2 `aktion1` und für `index==0` auch noch am Anfang `aktion0`. Wenn man nur bei einem geringen Teil der `case`s das `break` absichtlich weglässt, ist es sinnvoll, dieses „Durchfallen“ (*fall-through*) auf den nächsten `case` durch einen Kommentar[2] zu kennzeichnen, damit später nachvollziehbar ist, ob und wenn ja welche `break`-Anweisungen vergessen wurden.

Die `switch`-Anweisung akzeptiert nur für Ganzzahl-Ausdrücke (dazu gehören in ***C*** auch Zeichen). `case`-Sprungmarken dürfen nur konstante Ganzzahl-Ausdrücke enthalten, d.h. `case meinevariable` ist nicht möglich.

## 2.3. Schleifen

Für Aufgaben, die wiederholt ausgeführt werden, bieten sich Schleifen an. ***C*** bietet hier drei mögliche Konstruktionen: `while`-, `for`- und `do...while`-Schleifen. Die ersten beiden sind so genannte abweisende Schleifen, die durchlaufen werden, wenn eine bestimmte Bedingung erfüllt ist, letztere ist eine annehmende Schleife, die die Schleife durchläuft und anschließend eine Nicht-Abbruch-Bedingung testet (also mindestens einmal die Schleife durchläuft).

### 2.3.1. `while`

Als Beispiel soll hier die Ausgabe der ersten $n$ Fibonacci-Zahlen[3] dienen, die durch die Rekursion $a_{n+1} = a_n + a_{n-1}$, $a_0 = a_1 = 1$, definiert sind. Die `while`-Anweisung wertet die folgende Bedingung aus und führt einen Durchlauf des folgenden Anweisungsblocks bzw. der folgenden Anweisung durch. Syntax: *while (Bedingung) Anweisung;*

```
int main (void)
{
```

[2] Im Englischen hat sich hierfür `/* FALLTHRU */` eingebürgert.

[3] Benannt nach LEONARDO VON PISA, bekannt als FIBONACCI, der in Europa das Rechnen im „arabischen“ Dezimalsystem bekanntmachte. Von ihm stammt die Fibonacci-Folge, die aus einem – sehr einfachen – Modell zur Kaninchen-Vermehrung (alle Kaninchenpaare gleich, unsterblich, monogam und nicht inzuchtgefährdet) entsteht: Ein Kaninchenpaar bekommt ab dem zweiten Lebensmonat jeden Monat ein anderes Paar als Nachwuchs. Damit berechnet sich die Zahl von Kaninchenpaaren im Monat $n+1$ zu $a_{n+1}$ = *alle bisher lebenden Paare, d.h.* $a_n$, *plus die Nachkommen der Paare, die mindestens zwei Monate alt sind, d.h.* $a_{n+1-2}$

```
    int aktuelle, letzte, vorletzte, i, n;

    // n einlesen
    printf("Geben Sie n>=2 ein: ");
    // Bei Erfolg gibt scanf Anzahl zurueck
    if (scanf("%d", &n)!=1) {
        printf("Fehler: n konnte nicht gelesen werden\n"
               "... verwende stattdessen %d\n", n=30);
    }

    // Initialisierung
    letzte = vorletzte = 1;
    printf("a_1\t= %d\n", letzte);

    i = 2;
    while (i<=n) { // solange i<=n ist:
        // berechne a_i, gib es aus
        aktuelle = letzte + vorletzte;
        printf("a_%d\t= %d\n", i, aktuelle);
        // Inkrement
        vorletzte = letzte;
        letzte = aktuelle;
        i = i+1;
    }
}
```

Das Zeichen `\t` steht für einen Tabulator, d.h. den Sprung zur nächsten Tabellen-Zeichenposition; diese ist auf den meisten Systemen in der Regel nach jeder achten Zeichenposition. Das Format `%d` bezeichnet bei `printf` und `scanf` einen `int`-Wert (wobei `scanf` auch Werte in anderen Zahlensystemen akzeptiert). `scanf` gibt zurück, wie viele von den angeforderten Objekten eingelesen werden konnten; stimmt die Zahl (in diesem Fall 1) nicht, liegt ein Fehler vor. Die Zuweisung des Wertes 1 an `letzte` und `vorletzte` durch eine mehrfache Zuweisungs-Anweisung ist zulässig und wird zuweisungsweise von rechts nach links abgearbeitet; d.h. es wird zunächst 1 ausgewertet und dann `vorletzte` zugewiesen und anschließend der Wert, der rechts der zweiten Zuweisung steht (also wiederum 1), `letzte` zugewiesen.

### 2.3.2. `do while`

Eine `do`…`while`-Schleife hat die Syntax *do Anweisung while Bedingung;* und führt dementsprechend erst einmal die Anweisung oder den Anweisungsblock aus, bevor die Bedingung getestet wird.

```
    // Initialisierung
    letzte = 1; vorletzte = 0;
    i = 1;
```

```
do { // solange i<=n ist:
    // berechne a_i, gib es aus
    aktuelle = letzte + vorletzte;
    printf("a_%d\t= %d\n", i, aktuelle);
    // erhoehe i
    vorletzte = letzte;
    letzte = aktuelle;
    i = i+1;
} while (i<=n);
```

Die obligatorische Ausgabe von `a_1` erfolgt nun automatisch beim ersten Mal. Wichtig ist das Semikolon nach der `while`-Anweisung.

### 2.3.3. `for`

In diesem Beispiel gibt es eine Initialisierung sowie ein Auffrischen der Variablen `i`, `letzte` und `vorletzte`. Die `for`-Schleife kommt diesem Verhalten entgegen, indem sie neben der Bedingung noch Raum für die Initialisierung und Anweisungen, die nach jedem Schleifendurchlauf durchgeführt werden sollen, bietet. Syntax: *for (Initialisierung; Bedingung; Inkrement) Anweisung;* In der Regel beschränkt man sich hierbei auf eine Schleifenvariable:

```
// Initialisierung
letzte = vorletzte = 1;
printf("a_1\t= %d\n", letzte);

                // Inkrement von i
for (i=2; i<=n; i=i+1) { // solange i<=n ist:
    // berechne a_i, gib es aus
    aktuelle = letzte + vorletzte;
    printf("a_%d\t= %d\n", i, aktuelle);
    // Inkrement von vorletzte, letzte
    vorletzte = letzte;
    letzte = aktuelle;
}
```

Es ist aber ohne weiteres möglich, mit Hilfe von Kommata eine Reihe von Anweisungen sowohl für die Initialisierung als auch für das Inkrement anzugeben:

```
printf("a_1\t= %d\n", 1);

// Initialisierung
for ( letzte = vorletzte = 1, i=2; i<=n;
      i=i+1, vorletzte=letzte, letzte=aktuelle)
    printf("a_%d\t= %d\n", i,
              aktuelle = letzte + vorletzte);
```

Diese Konstruktion ist natürlich nicht in jedem Fall übersichtlicher. Manchmal hilft es bei längeren Schleifen aber, wenn man weiß, was vorher und nach jedem Durchlauf passiert. Das Verschieben

der Zuweisung in die Argumentliste von `printf` ist zulässig: Zunächst wird die Zuweisung abgearbeitet, dann wird das Ergebnis an `printf` übergeben.

### 2.3.4. Schleifen und Felder

Sollen in obigem Beispiel etwa die ersten 100 Fibonacci-Zahlen nicht nur berechnet und ausgegeben werden, sondern auch noch weiterverwendet werden, müssen sie irgendwo gespeichert werden. Dazu verwendet man Felder (*Arrays*). Die Deklaration eines Feldes sowie den Zugriff auf seine Elemente erkennt man an den eckigen Klammern:

```
int feld1[20];            // Integerfeld mit 20 Eintraegen
int feld2[4] = {0,1,2,3}; // Deklaration und Definition
                          // eines Integerfelds mit
                          // 4 Eintraegen.
int feld3[] = {4,5,6,7};  // Dito; das Zaehlen der Anzahl
                          // der Feldelemente uebernimmt
                          // der Compiler
```

Die Zuweisung der Feldwerte en bloc ist nur bei der Deklaration[4] eines Feldes möglich. Falls die Zahl in eckigen Klammern und die Anzahl an zugewiesenen Elementen nicht zusammenpasst, gibt der Compiler auf Wunsch eine Warnung aus und verfährt so, dass er ein so großes Feld anlegt, wie links von der Zuweisung angegeben ist und das Feld dann mit den Werten auf der rechten Seite füllt, soweit er kommt (also bis zum Minimum aus Feldgröße und Anzahl der Initialisierungswerte). Einzelne Feldelemente sind von dem Datentyp, von dem ein Feld angelegt wurde. Auf sie kann mittels des Index zugegriffen werden:

```
int feld2[4] = {0,1,2,3}, temp, index, i;

index = 1;
....
temp = feld2[1];
for (i=0; i<4; i++) { // Felder starten bei 0
    feld2[i] /= 3;
}
feld2[index] = temp;
```

Wie im Code angemerkt, liegt das erste Element des Feldes beim Index 0, das letzte also beim Index *Feldgröße*$-1$, siehe auch Abbildung 2.1. Falsche Laufbereiche für Feldindizes in Schleifen gehören zu den „beliebtesten“ Fehlerquellen in ***C***.

$N$-dimensionale Felder, etwa zweidimensionale für Matrizen, werden als Felder von $N-1$-dimensionalen Feldern definiert:

```
double id_2[2][2] = {{1.0,0.0},{0.0,1.0}};
```

[4] Es ist möglich, dass eine Variable an anderer Stelle im Programm bereits bekanntgemacht wurde, aber an dieser Stelle erst „richtig“ deklariert wird, d.h. so, dass die Variable auch im Speicher angelegt wird. Siehe hierzu auch die Erklärung des Schlüsselwortes `extern` im Abschnitt 4.15. Griffiger formuliert: Nur bei einer „Initial“-Deklaration ist eine „Initial“-Definition möglich.

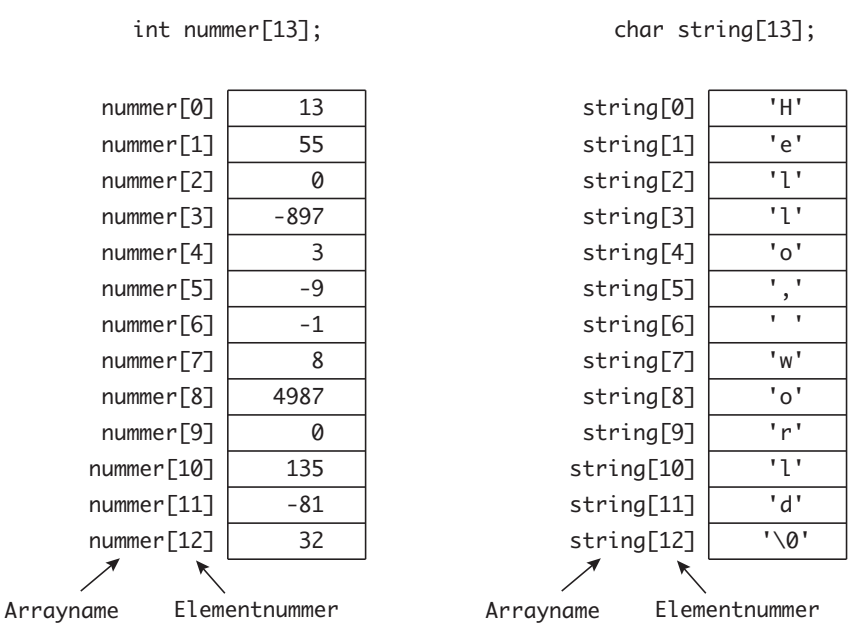



**Abbildung 2.1.:** Speicherdarstellung von Feldern und Strings.

```
id_2[0][0] = 1/3.0; // skaliere x-Komp.
```

Hierbei ist zu beachten, dass nur der letzte Feldindex, d.h. derjenige, der am nächsten beim Feldnamen steht, bei der Deklaration ausgelassen werden darf:

```
double id_2[][2] = {{1.0,0.0},{0.0,1.0}};
```

ist gültig, aber

```
double id_2[][] = {{1.0,0.0},{0.0,1.0}};
double di_2[2][] = {{0.0,1.0},{1.0,0.0}};
```

sind falsch.

„Normale“ Zeichen sind in ***C*** vom Typ `char` und werden in IndexHochkommata (einfachen Anführungszeichen) geschrieben, z.B. `'a'`. Strings aus solchen Zeichen sind *Character-Arrays*, wobei der String durch das Zeichen mit dem Wert 0 terminiert wird – und werden muss! `char` ist ein Ganzzahl-Datentyp, der genau ein Byte[5] umfasst und in der Regel einen kleineren Zahlenbereich umfasst als `int`. Häufige Fehler im Umgang mit Strings sind die fehlende Terminierung durch 0 und das Vergessen der Tatsache, dass ein String immer höchstens ein Zeichen weniger speichern kann als ein `char`-Array gleicher Größe:

[5] Jede Speicherzelle des Rechners enthält ein Byte; da die entsprechenden Schaltkreise nur „ein“ und „aus“ kennen, besteht ein Byte aus einer bestimmten Anzahl von Ein/Aus- bzw. 1/0-Informationen (Binärziffern, *Bits*). Bei den meisten heute im Privatbereich verwendeten Rechnern enthält ein Byte acht Bits, kann also $2^8 = 256$ verschiedene Zustände speichern. Der Datentyp `char` umfasst also je nach Maschine unterschiedlich große Alphabete – in der Regel sind diese zu klein für die Menge der benötigten Zeichen. Um diesem Umstand abzuhelfen, gibt es Multibyte-Zeichen sowie den Datentyp `wchar_t`, der eine maschinenunabhängige Alphabetgröße aufweist.

```
char zeichen1, zeichen2 = '0';
char zeichenfeld[6] = {'H','e','l','l','o'}; //kein String
char string[] = {',',' ','w','o','r','l','d','\n',0};

zeichenfeld[5] = '\0'; // Jetzt ist zeichenfeld ein String
```

Dieser Sachverhalt ist auch in Abbildung 2.1 dargestellt. Das Zeichen `'\0'` hat den Wert 0 und wird meistens statt des Wertes verwendet, um klarzumachen, dass es sich um den Stringterminator handelt. Diese Art, Strings bei der Deklaration zuzuweisen, ist ziemlich umständlich, deshalb kann man auch direkt den String angeben:

```
char string[] = ", world\n";

printf("%d",string[8]); //gibt 0 aus
```

Nach der Deklaration arbeitet man wieder elementweise mit `char`-Arrays, also auch mit Strings; das ist natürlich vergleichsweise umständlich. Deshalb gibt es Funktionen, die den Umgang mit Strings erleichtern, beispielsweise `strcpy()`, `strcat()`, `strcmp()`, `strlen()`. Diese und andere sind in `<string.h>` deklariert.

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

....

char string[80] = "Hello"; // string[0] bis [5] belegt
char hilfsstring[20];
int len;

len = strlen(string); // Laenge des Strings ohne '\0'
strcpy(hilfsstring,", world\n"); // kopiert das zweite
                                 // String-Argument in
                                 // das erste.
strcat(string, hilfsstring); // haengt den zweiten String
                             // an den ersten an; dabei
                             // wird der String-Termina-
                             // tor 0 des ersten Strings
                             // ueberschrieben, d.h. der
                             // Feldeintrag string[len]
                             // ist ','.
if ( strcmp(string, "Tach, Welt!\n") == 0 )
    printf("Die Strings sind gleich.\n");
else {
    printf("Die Strings sind nicht gleich; es ist");
    printf(" \"%s\" herausgekommen.\n",string);
}
```

Zu beachten ist bei `strcpy` und `strcat`, dass genügend Platz für den zu kopierenden bzw. anzuhängenden String in dem Feld enthalten sein muss, und bei `strcmp`, dass die Strings bei der Rückgabe 0 gleich sind. Die Angabe nur des Stringnamens entspricht einer Referenz auf den Feldanfang, d.h. `string` ist identisch zu `&string[0]`. Referenzen werden beispielsweise auch für `scanf` gebraucht. Das Format für Strings in `printf` und `scanf` ist `%s`, das für einzelne Zeichen `%c`. Auch hier gilt zu beachten, dass `'a'`$\neq$`"a"`.

Ursprünglich ist in ***C*** die Feldgröße eine Konstante, die zur Kompilierzeit festgelegt wird. In ***C***99 ist es möglich, die Feldgröße dynamisch anzugeben:

```
void DynFeld (int dim)
{
    int feld[dim];

    ....
}
```

Die dynamische Speicher-Allokation von ***C*** wird ebenfalls bei den Zeigern besprochen.

In den Abschnitten 4.7.3 und 6.7 werden Felder bzw. Felder als Parameter von Funktionen besprochen.

### 2.3.5. `break` und `continue`

Manchmal gibt es zusätzliche Kriterien für den Abbruch, die erst im Laufe des Anweisungsblocks testbar werden; hierfür wird die `break`-Anweisung verwendet. Andererseits gibt es auch Fälle, in denen nur der Anfangsteil der Schleifenanweisungen notwendig ist. Der Rest könnte dann durch entsprechende `if .... else`-Konstruktionen ausgeklammert werden. Als Alternative gibt es die `continue`-Anweisung, die direkt zum neuen Schleifendurchlauf springt. Sollen nur die durch 3 teilbaren Fibonacci-Zahlen unter 1000 (aber höchstens bis $a_n$) ausgegeben werden, ist das bei einer Umstellung beispielsweise des `for`-Codes leicht möglich:

```
// Initialisierung
printf("a_1\t= %d\n", 1);

for ( letzte = vorletzte = 1, i=2; i<=n;
      i=i+1, vorletzte=letzte, letzte=aktuelle)
{
    aktuelle = letzte + vorletzte;
    if (aktuelle >= 1000)
        break;
    if (aktuelle%3)
        continue;
    printf("a_%d\t= %d\n", i, aktuelle);
}
```

Die Bedingung `(aktuelle%3)` ist ein Beispiel für typischen ***C***-Code unter Ausnutzung des Ganzzahl-Charakters der Wahrheitswerte: Der *modulo*-Operator `%` liefert den Rest der Ganz-

zahldivision (d.h. `0%3`= 0, `1%3`= 1, `2%3`= 2, `3%3`= 0, `4%3`= 1, `-1%3`= 2), die Teilbarkeit durch 3 bedeutet einen Rest von 0 modulo 3, führt also zu der Bedingung `(aktuelle%3)` $\neq 0$. Nachdem aber gerade nur `0` als falsch interpretiert wird, kann der Vergleich mit 0 weggelassen werden. Will man ihn ausschreiben, verwendet man den Vergleichsoperator `!=`, d.h. der Wahrheitswert von `((aktuelle%3) != 0)` wird getestet.

### 2.3.6. Spezialitäten bei `for` und Inkrement

Man kann die `for`-Schleife variieren, indem man Initialisierung, Bedingung oder Inkrement weglässt. Beispielsweise ist `for (;i<5;)` äquivalent zu `while (i<5)` und `for (;;)` eine Endlosschleife[6].

Bei der Initialisierung der `for`-Schleife gilt es zu beachten, dass in ANSI-***C*** in der Regel nur Variablen gleichen Typs ein Wert zugewiesen werden sollte. Diese Einschränkung gibt es in ***C***99 nicht mehr. Zudem können hier Schleifenvariable, die nur für die Dauer der Schleife gebraucht werden, im ***C***++-Stil definiert werden:

```
for ( int i=2; i<=n; i+=1)
{ // Hier lebt i
    ....
} // Hier nicht mehr
```

In ***C*** gibt es neben dem typischen Zuweisungsoperator `=` noch andere Zuweisungsoperatoren, die auf alle arithmetischen (und bitweisen) binären Operationen anwendbar sind: Wenn der erste Operand gleich der Zielvariable ist, kann auch die Schreibweise *Zielvariable (Operator)= zweiterOperand* verwendet werden. So ist die obige Anweisung `i+=1` äquivalent zu `i=i+1`. Die Zuweisungsoperatoren `-=`, `*=`, `/=` und `%=` arbeiten in gleicher Weise. Hierbei handelt es sich nicht unbedingt nur um Schreibweisen; unter Umständen entsteht auch unterschiedlicher Maschinencode[7].

Gerade für `i+=1` geht es aber noch kürzer und liefert uns eine weitere ***C***-typische Spezialität: Die *Inkrement*- und *Dekrementoperatoren*. Diese erhöhen bzw. erniedrigen einen Datentyp auf den nächsten logisch sinnvollen Wert, also im Falle von Ganzzahl-Datentypen auf den um 1 größeren bzw. kleineren Wert; zudem spielt bei ihnen die Ausführungsreihenfolge eine Rolle. Im Falle des Präinkrements/dekrements `++i`/`--i` wird der Wert um 1 verändert und dann zurückgeliefert, im Falle des Postinkrements/dekrements `i++`/`i--` wird der aktuelle Wert zurückgeliefert und dann um 1 verändert:

```
int a, b, c;

// Schleife aequivalent zu oben
for ( int i=2; i<=n; i++)
```

[6] Oftmals wird ein Makro `FOREVER` als `for (;;)` definiert; eine andere Alternative zum Markieren einer Endlosschleife ist `while (1)`.

[7] Auf manchen CISC-Prozessoren dauert etwa das Laden von zwei Variablen plus anschließendes Addieren, Subtrahieren usw. einen Taktzyklus länger als das Laden und Addieren, Subtrahieren usw. nur einer Variable. Gute Compiler „wissen“ das aber und produzieren auch bei der langen Schreibweise den kurzen Code. Somit bleibt es der Schreibfaulheit des Programmierers überlassen, ob er diese Variante bevorzugt.

```
{ // Hier lebt i
    ....
} // Hier nicht mehr

a = 0; b = 1;
c = (++a)*(++b); // a=1; b=2; c=1*2;
c = (a--)+(b++); // c=1+2; a=0; b=3;
```

Die Inkrement- und Dekrementoperatoren sind nicht nur eine praktische Kurzschreibweise für eine Veränderung des Wertes um 1, sondern in der Regel für spezielle Zwecke auch direkt in der Hardware vorhanden[8], also unter Umständen schneller.

[8]Typisch ist für ältere Prozessoren etwa die Paarung aus Postinkrement und Prädekrement für das Ablegen und Wiederaufnehmen von Registerwerten über den Stack. Zudem besteht die Möglichkeit, dass Inkrement- und Dekrementoperationen im gleichen Taktzyklus ausgeführt werden wie die Operation, bei der sie zur Anwendung kommen. Für Ganzzahl-Datentypen ist es ebenfalls wieder eine Sache des Compilers, was übersetzt wird. Bei Zeigern (siehe 4.7) sind die Operatoren dagegen fast unerlässlich.

***Programmlisting 2.1:*** polynom-interpolation.c

```
/*****************************************************************
**
** polynom-interpolation -- Baue eine Polynom, das vorgegebene
**                          Stuetzpunkte interpoliert und berechne
**                          Wert an Stelle x=0 zu geg. Genauigkeit.
**
** Kompilierung: zusaetzliche Compileroption -lm (linke Mathe-Bib.
**               libm.a hinzu; lib und .a werden immer weggelassen)
**               Notwendig wegen Verwendung von fabs().
**
** Geschichte:
**
**             2004-04-16: Kleinere Aenderungen durch M. Mair
**
**             2004-04-15: Neues Leben eingehaucht durch J. Werner
**
**             Finstere Vergangenheit: Irgendwo abgestaubt
**
*****************************************************************/


/*** INCLUDES & DEFINES ******************************************/

#include <stdio.h>
#include <math.h>   // fuer die Absolutwertfunktion fabs()

/* kann in jeder Funktion der Datei als konstanter Wert benutzt
** werden; Alternative:
** #define MAX 20
*/
const int MAX = 20;


/*** Funktionsprototypen *****************************************/



/*****************************************************************
**
** Hauptprogramm
**
** Extrapolation/Interpolation auf 0
** Algorithmus: Schema von Aitken-Neville (zur Auswertung des
**              Polynoms an nur wenigen Stellen).
**              Grundidee:
**         P_(i_0..i_k)(x)=
**            (P_(i1..i_k)(x)(x-x_(i_0)-P_(i0..i_k-1)(x)(x-x_(i_k)))/
**                        (x_(i_k)-x_(i_0)),
**              wobei x_(i_j) die x-Werte der y-Stellen sind und
**              P_(i_j1..i_j2)(x) der Wert des Interpolationspolynoms
**              durch die Stuetzstellen (x_(i_j1), y_(i_j1)) bis
```

```
**               (x_(i_j2), y_(i_j2)) an der Stelle x ist und
**               P_(i_j)(x) = y_(i_j)
**
** Argumente: Keine
**
** Rueckgabe: 0 .. kein Fehler
**
*****************************************************************/

int main (void)
{
    int n;                      // Anzahl der Wertepaare
    double x[MAX], y[MAX];      // Wertepaare
    int i, j;                   // Schleifenvariablen
    double eps;                 // Genauigkeit
    double P[MAX];              // Polynom-Werte bei 0
    double q;
    int genau = 0;              // gewuenschte Genauigkeit erreicht

    printf("Beispielprogramm zur Extrapolation\n");

    // Einlesen der Stuetzstellen
    printf("Bitte geben Sie die Anzahl der Stuetzstellen an: ");
    scanf("%d", &n);
    n--; // n um 1 erniedrigen

    for (i = 0; i <= n; i++) {
        printf("%d. Stuetzstelle\n", i);
        printf("\tBitte geben sie den x-Wert an: ");
        scanf("%lf", &x[i]);
        printf("\tBitte geben sie den y-Wert an: ");
        scanf("%lf", &y[i]);
    }

    // Einlesen der Genauigkeit
    printf("Bitte geben Sie die gewuenschte Genauigkeit an: ");
    scanf("%lf", &eps);

    /* Berechnung der Koeffizienten nach Aitken-Neville Schema:
    ** zeilenweise von oben nach unten, d.h. unter Hinzufuegen
    ** von jeweils nur einer Stuetzstelle.
    ** Speziell: Abbruch, wenn zwei aufeinanderfolgende
    ** Approximationen einen relativen Fehler von eps aufweisen.
    */
    for (i = 0; i <= n; i++) {
        q = y[i];

        for (j = 1; j <= i; j++) {
            double r = q - P[j - 1];

            P[j - 1] = q;
            q = P[j - 1] + x[i] * r / (x[i - j] - x[i]);

            if (fabs(q - P[j - 1]) < eps * fabs(P[j - 1])) {
                /* Genauigkeit liegt im erlaubten Bereich */
                genau = 1;
```

```
                break; // verlasse die for-Schleife
            }
        }
        if (genau)
        break;
        P[i] = q;
    }

    // Ausgabe des Polynomwerts
    printf("\nWert des Interpolationspolynoms p(0) = %f\n", q);

    return(0);
}
```

***Programmlisting 2.2:*** runge-kutta.c

```
/*****************************************************************
**
** runge-kutta -- naeherungsweise Loesung eines Anfangswertproblems
**
** Kompilierung: zusaetzliche Compileroptionen -lm -std=c99
**               ohne letzteres funktioniert for (int i=... nicht.
**
** Geschichte:
**             Liest eh keiner (falsche Einstellung)
**
*****************************************************************/


/*** INCLUDES & DEFINES ******************************************/

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


/*** Funktionsprototypen *****************************************/

double f (double x, double y);
double exakteLoesung (double x);


/*****************************************************************
**
** Hauptprogramm
**
** Wir loesen y'=f(x,y), y(x_0)=y_0.
** Algorithmus: klassisches Runge-Kutta-Verfahren; Informationen:
**              Numerik-Buch oder -Vorlesung Ihres Vertrauens.
**              Grundidee: Wir laufen von Stuetzstelle zu Stuetzstelle
**              und nutzen dabei die Ableitungsinformation f aus.
**
** Argumente: Keine
**
** Rueckgabe: 0 .. kein Fehler
**
*****************************************************************/
```

```
int main (void)
{
    double xi;    // aequidistante Stuetzstellen
    double yi;    // Naeherungswerte des Polygonzugverfahrens
    double h;     // Abstand der Stuetzstellen
    double C=1.0; // Faktor fuer Lsg.
    int n;        // Anzahl der Stuetzstellen
    // Hilfsvariablen
    double k1, k2, k3, k4, k;

    printf("Beispielprogramm zum Runge-Kutta-Verfahren\n");
    // Werte einlesen
    printf("Bitte den Startwert x0 eingeben: ");
    scanf("%lf", &xi);
    printf("Bitte die Anfangsbedingung y(%f) eingeben: ", xi);
    scanf("%lf", &yi);
    printf("Bitte Anzahl der Stuetzstellen eingeben: ");
    scanf("%d", &n);
    printf("Bitte den Abstand der Stuetzstellen eingeben: ");
    scanf("%lf", &h);

    // die Ergebnisse sollen tabellarisch ausgegeben werden
    printf("      xi      yi (exakt)    yi(Runge-Kutta)\n");

    // Faktor berechnen
    if ( (C=exakteLoesung(xi)) == 0.0 ) {
        if ( yi != 0.0 )
            printf("x0 und y0 passen nicht zusammen!\n");
    }
    else
        C = yi/C;

    /* wiederhole fuer jede Stuetzstelle
    obere Grenze wird mit (n + 1) angegeben, damit auch die n-te
    Stuetzstelle berechnet wird */
    for (int i = 0; i < (n + 1); i++) {
        /* um eine tabellarische Ausgabe zu erhalten, ist bei den
        ** Formatangaben eine Breitenangabe der Variablenausgabe
        ** angegeben, also fuer xi eine Breite von 10 Zeichen, fuer
        ** die exakte Loesung eine Breite von 12 Zeichen und 15
        ** Zeichen fuer yi; zusaetzlich sollen nur drei
        ** Nachkommastellen angezeigt werden, deshalb ist die
        ** Praezision .3 angegeben.
        */
        printf("%10.3f %12.7g %15.7g\n", xi, C*exakteLoesung(xi), yi);

        // Runge-Kutta-Verfahren
        k1 = f(xi, yi);
        k2 = f(xi + h / 2.0, yi + k1 / 2.0);
        k3 = f(xi + h / 2.0, yi + k2 / 2.0);
        k4 = f(xi + h, yi + k3);
        k = (k1 + 2.0 * k2 + 2.0 * k3 + k4) / 6.0;
        yi = yi + h*k;    // koennte auch lauten: yi += k;

        // naechste Stuetzstelle
```

```
        xi = xi + h;    // koennte auch lauten: xi += h;
    }

    return(0);
}

// Beispiel-DGL y' = f(x, y)
double f (double x, double y)
{
    return(x * y);
}

// berechnet die exakte Loesung der DGL
double exakteLoesung (double x)
{
    return(exp(x * x / 2.0));
}
```

# 3. Präprozessor und Compiler

640K ought to be enough for anybody.

*(wird Bill Gates zugeschrieben)*

## 3.1. Präprozessor

### 3.1.1. Makros

Wenn man jeweils eine Zeile als String einlesen will, kann man das mittels der Funktion `gets()` tun. `gets()` überprüft allerdings nicht, ob die Zeile in das Feld passt. Daher wählt man das Feld für den Zeilenpuffer in der Regel „groß genug“. Auch für andere Felder gilt ähnliches: Ursprünglich war man der Meinung, dass sie nie größer gebraucht werden könnten, also sind sie von einer festen Größe. Schreibt man diese feste Größe fest für alle Schleifen, Zugriffe, usw. ins Programm und es kommt doch eine Änderung, müssen alle Bezüge auf die *magic number*[1] geändert werden. Wesentlich eleganter ist es, eine symbolische Konstante zu definieren, die dann überall vorkommt. Eine Änderung der Konstanten reicht dann aus, um alle Bezüge auf das Feld zu ändern. Symbolische Konstanten und Funktionen (*Makros*) werden mit der `#define`-Anweisung des Präprozessors definiert:

```
   #define NAMELEN  (32)
   #define ARRAYDIM 100

   #define MULTADD(a,b) ((a)*(b)+23)
   #define MAX(a,b) ((a>=b)?(a):(b))
7  #define LANGESDEFINE(a,b,c) ( (a<1) ? ((b>=c) ? ((c>a) ?\
                                                    (b):(c+2))\
                                                   : (a*2))\
                                        : ((b%=c) ? (23):(1))\
                                  )
```

[1] Der Begriff *magic number* ist nicht a priori negativ. Eine Definition aller möglichen Bedeutungen im Hinblick auf die Programmierung findet sich etwa im *jargon file* (→ Jargon File, Online-Version des New Hacker's Dictionary, siehe z.B. `http://www.ccil.org/jargon/jargon.html`). Kurz gesagt sind mögliche Bedeutungen:
a) Im Quellcode: Nicht offensichtliche Konstanten, deren Wert signifikant für Funktionieren und Funktionsweise des Programmes ist und die „hart“ einprogrammiert sind (anstatt per symbolischer Konstante).
b) Zahl, die für einen Algorithmus kritische Informationen auf undurchsichtige Weise kodiert (ursprüngliche Bedeutung).
c) Spezielle Daten am Anfang einer Binärdatei, die die Bestimmung des Typs der Datei (Grafik-Datei in bestimmten Format o.ä.) ermöglichen.
d) Eingabe, die zu Grenzen bei Rechnungen führt, an denen unstetiges/inkonsistentes Verhalten auftritt (z.B. die Null bei der Division oder die 100 bei der zweistelligen Kodierung von Jahreszahlen).

```
int main (void)
{
    char name[NAMELEN+1];
    float allezahlen[ARRAYDIM];
    int i = 1, j = 2, k = 8;
    float f = 1.0, g = -20.0;

    printf("Maximum von i und j: %d", MAX(i,j));
    j = LANGESDEFINE(i,k,j);
    f /= MULTADD(f,g);

    fgets(name, NAMELEN+1, stdout); // besser als gets()
    if (name[strlen(name)-1] != '\n')
        printf("fgets-Fehler: Evtl. name zu klein\n");
    for (i=0; i<ARRAYDIM; i++)
        allezahlen[i] = 5.0/(((float)i)+1.0);
}
```

Die Verwendung der Konstanten ist selbsterklärend, die Verwendung der symbolischen Funktionen ist etwas undurchsichtiger. Alle `#define`-Anweisungen sind nur Textersetzungsregeln, die der Präprozessor ausführt, d.h. nach dem Durchlauf des Präprozessors steht in der Deklaration von `name` tatsächlich `char name[(32)+1];` beim „Aufruf“ von `MULTADD` in Wirklichkeit `f /= ((f)*(g)+23);` und im `printf`-Aufruf `((i>=j)?(i):(j))`. Letzteres ist ein Beispiel für den ternären *Conditional*-Operator `?:`. Dessen drei Argumente sind *(Bedingung)*`?`*(WertFallsWahr)*`:`*(WertSonst)*, die nach Art einer verkürzten `if .... else`-Anweisung bearbeitet werden. Die Klammern sind nicht (alle) notwendig, aber sinnvoll, eben weil es sich nur um Textersetzung handelt, um Missverständnisse auszuschließen:

```
#define APLUSB(a,b)  a+b

int main (void)
{
    int a = -1, b = 2, c;

    // gedacht: (a+b)*(a+b)=1
    c = APLUSB(a,b)*APLUSB(a,b);
    // tatsaechlich a+b*a+b=-1
    ....
}
```

Solche Fehler sind mitunter schwer zu finden, deshalb sind hier wie im Zweifelsfall immer mehr Klammern besser als zu wenige. Falls die `#define`-Anweisung keinen Wert zugewiesen bekommt, wird der Ausdruck als ungleich 0 interpretiert, in der Regel als $-1$ oder 1.

Makros können in ***C*** aus beliebigen Buchstaben und Zahlen bestehen, wie auch die normalen Bezeichner, allerdings werden meistens für Variablen Kleinbuchstaben und für Makros Großbuchstaben bevorzugt. Bei beiden verwendet man oft Unterstriche zur Trennung von Wörtern Bei Funktionsnamen und mehr noch bei komplexen Datentypen wie `struct`, `union` sowie eige-

nen Datentypen (siehe Abschnitte 4.9, 4.10, 4.17) werden die Einzelwörter manchmal am Anfang groß geschrieben und es wird auf Unterstriche verzichtet. Hierbei handelt es sich aber lediglich um Konventionen, die sich eingebürgert haben und je nach Community nicht einmal einheitlich sind. Als Faustregel für den Hausgebrauch kann gelten, dass die für den speziellen Zweck übersichtlichere Variante bevorzugt und konsistent durchgehalten werden sollte.

Präprozessor-Anweisungen sind immer genau eine Zeile lang; deshalb müssen Zeilenumbrüche wie bei Strings mit dem Backslash \ am Zeilenende vermieden werden und es dürfen keine zwei Anweisungen in einer Zeile stehen.

Der Präprozessor hat noch andere Aufgaben, die im Prinzip alle unter dem Oberbegriff Textersetzung laufen:

- Kommentare entfernen: Alle Kommentare werden durch nichts ersetzt.
- Zeilenumbrüche `\n` mit vorangestelltem \ durch nichts ersetzen, d.h. alles in eine Zeile schreiben.
- Alle mit `#include`-Anweisungen eingebundenen Dateien an Ort und Stelle einfügen.
- Bedingte Kompilierung: Auswertung von `#if`-, `#ifdef`-, `#elif`-, `#else`-, `#ifndef`-, `#endif`-, `#undef`-Anweisungen und Weglassen des durch die Bedingung ausgeschlossenen Codes.
- Makroersetzung in der Reihenfolge der `#define`-Anweisungen.
- Ersetzen/Auswerten der vordefinierten Bezeichner `__LINE__` und `__FILE__` (sowie in *C*99 `__func__`) sowie Durchführen der `#line`-Anweisung.
- Weiterreichen der `#pragma`-Anweisungen an den Compiler.

### 3.1.2. `#include`

Mittels `#include` werden die `.h`-Dateien (*Header-Dateien*) eingebunden, die ihrerseits Funktionsprototypen und Makro- und Typ-Definitionen zum Gebrauch der Funktionen enthalten. Sie sind notwendig, um die entsprechenden Funktionen verwenden zu können. Wenn man eigene Programme in mehrere Module zerlegt, ist es sinnvoll, jedem Modul eine Header-Datei zuzuordnen, die von anderen Modulen eingebunden wird und die entsprechenden Prototypen und Definitionen enthält. Ein Modul besteht hierbei aus einer oder mehr Übersetzungseinheiten, denen in der Regel ebenfalls eine eigene Header-Datei zugeordnet ist. Die Modul-Header-Datei schließt dann ggf. nur alle Header-Dateien des Moduls ein.

Es ist prinzipiell möglich, beliebige Dateien mit `#include` einzubinden, also etwa auch Stücke von Code, die nur Funktionen oder Funktionsteile enthalten. Das kann manchmal sinnvoll sein, birgt aber immer die Gefahr, dass Dinge wie vergessene Klammern aus der eingebundenen Datei seltsame Auswirkungen auf den Code zeitigen.

Header-Dateien im System- oder Compiler-Include-Pfad, etwa die Header-Dateien zur Standard-Bibliothek werden per Dateiname geklammert durch die spitzen Klammern < und > eingebunden.

Header-Dateien, die sich im gleichen Verzeichnis befinden wie die Datei, in die sie eingefügt werden, oder in einem dem Compiler bekannt gemachten Include-Pfad, werden in Anführungszeichen gesetzt[2].

```
#include <stdio.h>

#include "meinheader.h"
```

### 3.1.3. `#line` und vordefinierte Makros

Zur Erleichterung der Fehlersuche und Ausgabe von Informationen, wo die Fehlersuche aufgetreten ist, gibt es die vordefinierten Bezeichner `__LINE__` und `__FILE__`, sowie in ***C***99 `__func__`: `__LINE__` enthält die aktuelle Zeilennummer in der entsprechenden Datei, deren Name in `__FILE__` enthalten ist. `__func__` enthält den Namen der aktuellen Funktion. Mittels der `#line`-Direktive, kann man die Werte von `__LINE__` und `__FILE__` verändern:

```
// Zeilenzaehler auf 100 setzen
#line 100
// Zeilenzaehler auf 0 setzen und mit der neuen|Zeile 100
// "Datei" NeuerAbschnitt.c starten.           |Zeile 101
#line 1 "NeuerAbschnitt.c"


int main (void)                                 // Zeile   3
{                                               // Zeile   4
    printf("%s: %d\n",__FILE__,__LINE__);       // Zeile   5
}                                               // Zeile   6
```

Neben diesen vordefinierten Bezeichnern, gibt es noch `__DATE__` und `__TIME__`, die den Zeitpunkt der Kompilierung als String enthalten. Der Wert `__STDC__` gibt an, ob es sich um einen Standard-***C***-konformen Compiler handelt oder nicht ($-1$ bzw. 0). Unter ***C***99 gibt es noch `__STDC_HOSTED__` (Implementierung akzeptiert alle strikt standardkonformen Programme, 1, oder akzeptiert alle standardkonformen Programme, die keine komplexen Datentypen verwenden und sich nur auf bestimmte Standard-Header[3] stützen, 0), `__STDC_VERSION__` (***C***-Version, mindestens 199901) sowie Makros, um die Erfüllung gewisser Spezifikationen anzuzeigen. Viele Compiler definieren eigene Makros, die unter Umständen ähnlich wie die von ***C***99 funktionieren.

### 3.1.4. Compiler-Direktiven mittels `#pragma`

Mit `#pragma`-Anweisungen kann man dem Compiler mitteilen, ob bestimmte Annahmen getroffen werden sollen, die z.B. beeinflussen, wie der ***C***-Code in Maschinensprache umgesetzt werden

[2] Wird ein in Anführungszeichen gesetzter Dateiname nicht im aktuellen Verzeichnis oder einem erweiterten Include-Pfad gefunden, so wird die `#include`-Anweisung behandelt, als stände der Name in spitzen Klammern.

[3] Diese Minimal-Header sind für ***C***99: `<float.h>`, `<iso646.h>`, `<limits.h>`, `<stdarg.h>`, `<stdbool.h>`, `<stddef.h>` und `<stdint.h>`.

soll. Auf manchen Rechnern gibt es für bestimmte Aufgaben eigene Hardware, deren Verwendung man dann etwa über `#pragma` ein- bzw. ausschalten kann. *C*99 definiert Direktiven für die Behandlung von Arithmetik mit Fließkomma- und komplexen Zahlen. Details über unterstützte Direktiven finden sich in der Compiler-Dokumentation.

### 3.1.5. Bedingte Kompilierung

Für größere Projekte wünscht man sich in der Entwicklungsphase und zur Wartung oft einen Testmodus, in dem es mehr Ausgaben gibt, zusätzliche Konsistenzchecks durchgeführt werden oder ähnliches. Eine Möglichkeit besteht darin, dem Programm von außen einen Parameter zu geben, der die Tests aktiviert. Andererseits vergrößern Tests, die nur für den Entwickler von Interesse sind, das fertige Programm bzw. die fertige Bibliothek nur unnötig. Unter Umständen verlangsamt sich das Programm auch erheblich, wenn viele `if`-Abfragen in Schleifen für den Testmodus erforderlich sind.

Mit bedingter Kompilierung durch Präprozessor-Anweisungen ist es möglich, zur Kompilierzeit festzulegen, ob das Programm im Testmodus laufen soll, wodurch man sich die Auswertung von `if`-Anweisungen spart und nur dann der Test-Code im Programm enthalten ist, wenn es tatsächlich getestet wird. Der folgende Code zeigt die Anwendung von `#if`, `#ifdef`, `#ifndef`, `#else`, `#elif`, `#endif` und `#undef`:

```
// definiere TEST, d.h. setze TEST != 0
#define TEST

// positiver Test
#ifdef TEST
#include "mytests.h"
#endif

....

int main (void)
{

   ....
 // Alternative
 #ifdef TEST
    berechnewas();
    gibesaus();
 #else
    printf("%s: %d\n",__FILE__,__LINE__);
 #endif
    ....
 // negativer Test
 #ifndef TEST
    machwasnichtzutestendes();
 #endif
```

```
    ....
  // Ab hier ist TEST nicht mehr definiert (also 0)
  #undef TEST
  // Mehrfach-Alternative
  #if defined LANGERTEST
     ....
  #elif defined MITTLERERTEST
     ....
  #elif defined KURZTEST
     ....
  #elif !defined NOTEST
     ....
  #endif
 }
```

Das Schlüsselwort `defined` erleichtert das Abtesten in Mehrfachalternativen. Einzeln ist `#if defined` äquivalent zu `#ifdef` und `#if !defined` äquivalent zu `#ifndef`. Die `#undef`-Anweisung bewirkt das Gegenteil der `#define`-Anweisung ohne Wertzuweisung. Im obigen Beispiel waren `LANGERTEST` usw. nicht definiert. Es ist aber beispielsweise möglich, beim Kompilieren über die Kommandozeile Makronamen auf „definiert“ zu setzen, so dass im eigentlichen Programm nichts geändert werden muss.

Die bedingte Kompilierung wird für den sogenannten *safe include*-Mechanismus eingesetzt; wenn eine Header-Datei `headera.h`, die in eine Programmdatei `test.c` eingebunden wird, von einer anderen Header-Datei `headerb.h`, die ebenfalls in `test.c` eingebunden wird, per `#include "headera.h"` angefordert wird, ergibt sich ein Fehler, da vom Präprozessor `headera.h` zweimal eingefügt wird, die entsprechenden Definitionen und Deklarationen also zweimal vorkommen. Mittels `#define`, `#ifndef` und `#endif` kann man sich leicht dagegen absichern:

```
/*********************************************************
** headera.h -- includes und defines fuer modul_a       **
*********************************************************/

// Safe include
#ifndef HEADERA_H
#define HEADERA_H

// Eigentliches Headerfile
....

#endif
//Dateiende
```

Beim ersten Durchlauf der Anweisungen zwischen `#ifndef` und `#endif` wird `HEADERA_H` definiert, dadurch kann es keinen zweiten Durchlauf geben, der entsprechende Code findet sich also nur ein einziges Mal.

### 3.1.6. Präprozessor-Operatoren

Der Präprozessor kennt den *stringize*-Operator # und den *pasting*-Operator ##. Ersterer verwandelt den nachfolgenden Text in einen String, letzterer verknüpft zwei Argumente:

```
#include <stdio.h>
#define stringize(s) # s
#define makestring(s) stringize(s)
#define concatenate(a,b) a ## b

int main()
{
    int var12=32;

    printf(makestring(C ist seltsam und sonderbar));
    printf("\nund %s hat den Wert %d\n",
           makestring(concatenate(var,12)),
           concatenate(var,12));
}
```

Das zusätzliche Makro `makestring` ist notwendig, damit der Wert im Argument von stringize ausgewertet wird; würde man die `makestring`- durch `stringize`-Aufrufe ersetzen, so würde im ersten Fall das gleiche, im zweiten aber der String `"concatenate(var,12)"` statt `"var12"` herauskommen.

Die beiden Operatoren werden eher selten verwendet, ermöglichen aber die Behandlung von bestimmten Spezialfällen.

### 3.1.7. `#error`

So, wie man während der Programm-Laufzeit auf Fehler testen und gegebenenfalls das Programm abbrechen kann, so kann man auch dafür sorgen, dass bestimmte Quelltexte nicht übersetzt werden können, wenn bestimmte Bedingungen verletzt sind. Für diesen Zweck gibt es die Präprozessor-Anweisung `#error`; wann immer der Präprozessor auf eine Zeile stößt, die mit `#error` beginnt, wird die Zeile ausgegeben; der Kompilier-Vorgang gilt als nicht erfolgreich beendet.

```
#ifdef TEST
#   error "TEST-Modus funktioniert noch nicht"
#endif

#if (!defined __STDC_VERSION__) \
    || (__STDC_VERSION__<199901)
#   error "Nicht C99-faehiger Compiler"
#endif
```

## 3.2. Compiler

Der Compiler übersetzt den vom Präprozessor vorbereiteten Quellcode in Assemblersprache. Nachdem ***C*** eine Hochsprache ist, kann der ***C***-Code nicht direkt in der Hardware abgearbeitet werden, sondern muss erst entsprechend in Anweisungen umgewandelt werden, die der Rechner versteht. Assemblersprache ist vom Prinzip her nur eine etwas verständlichere Aufschlüsselung der eigentlichen Maschinensprache des Rechners. Ein guter Compiler ist in der Lage, die vorhandene Hardware nahezu optimal auszunutzen, während ein weniger guter Compiler sich etwa nur auf einen Teil der Maschinenbefehle des Rechners als Zielmenge beschränkt. Zudem kann ein guter Compiler auch Programm-Konstrukte teilweise syntaktisch anders auflösen als der/die Programmierende sie aufgeschrieben hat, um einen Geschwindigkeitsgewinn zu erzielen.

Dadurch muss man bei vielen Dingen nicht mehr so stark aufpassen, die früher angesichts weniger leistungsfähigerer Compiler zu erheblichen Geschwindigkeitsverlusten geführt haben. Diese Optimierungen sind aber nur lokal, für die globale Geschwindigkeit eines Programmes ist immer noch entscheidend, ob man einen guten Algorithmus verwendet.

Ein wesentlicher Aspekt des Compilers ist auch die Überprüfung, ob der Code standardkonform ist, sowie die Ausgabe entsprechender Fehlermeldungen und Warnungen angesichts falscher bzw. ungewöhnlicher Konstrukte. Gerade letzterer Aspekt ist wesentlich für das Programmieren-Lernen. Mitunter ist zwar tatsächlich beabsichtigt, was an der Stelle steht, die eine Warnung hervorruft, aber es gibt unter Umständen geeignetere (sprachkonforme) Sprachmittel, um den gewünschten Effekt zu erzielen. Gerade unter dem Gesichtspunkt, dass ein bestimmtes Programm nicht nur auf einem einzigen Rechner und unter einer bestimmten Version eines Betriebssystems lauffähig sein sollte, ist es sinnvoll, sich klar zu machen, woher die Warnung rührt und warum man sie ignorieren kann oder nicht ignorieren sollte. Fehler beim Kompilieren führen zum Abbruch. Oft ist der Fehler genau dort zu suchen, wo der Compiler einen Fehler anmahnt – manchmal ist die Ursache aber eine ganz andere (etwa eine vergessene Klammer oder ähnliches). Die Compiler-Dokumentation gibt näheren Aufschluss über die Bedeutung der ausgegebenen Fehlermeldungen und Warnungen.

Nach der Kompilierung wird der Assembler-Code durch einen Assembler in Maschinensprache umgewandelt, es entsteht ein so genanntes *object file*, das in der Regel auf „.o“ endet. Abschließend werden ein oder mehrere Objektdateien entweder zu einem lauffähigen Programm zusammengebunden oder in einer Bibliothek zusammengefasst. Eine oder mehrere Bibliotheken können wiederum mit Objektdateien zu einem Programm zusammengebunden werden. Es ist zu beachten, dass alle Symbole (Namen), die nicht nur in einer Datei verwendet werden, erst beim Zusammenbinden aufgelöst werden müssen, dann aber in eindeutiger Weise. In allen Objektdateien (und Bibliotheken) zusammen gibt es genau eine Funktion `main()`. Das Zusammenbinden erfolgt durch den Linker. Linker hängen eng mit dem Betriebssystem und dessen Arbeitsweise zusammen, weswegen Compiler-Umgebungen sich in der Regel beim Linken auf einen vorhandenen Linker, der auch vom Betriebssystem zur Verfügung gestellt sein kann.

Ein „beliebter“ Fehler im Umgang mit dem Linker ist, das Hinzulinken benötigter Bibliotheken (zum Beispiel der Mathematik-Bibliothek mittels `-lm`) zu vergessen. Die Reihenfolge der Bibliotheken beim Linken ist ebenfalls nicht egal! Die elementarsten Bibliotheken sollten am weitesten hinten stehen. Auf manchen Systemen können Bibliotheken auch dynamisch gelinkt werden, d.h. sie werden erst zur Laufzeit geöffnet und die entsprechenden Funktionen zugänglich gemacht; dann muss die Bibliothek natürlich für das Programm auffindbar sein.

In Tabelle 3.1 finden sich einige Optionen des Compilers *gcc* . Für den Kurs sind die Aufrufe

`gcc -Wall -std=c99 -g -o` *Programmname C-Datei-Name(n)*,

bzw.

`gcc -Wall -std=c89 -g -o` *Programmname C-Datei-Name(n)*,

gegebenenfalls erweitert um die Optionen `-O` und `-lm`, sinnvoll.

## 3.3. Modularisierung

Die programmiererische Lösung eines Problems besteht in der Regel im Zerlegen des Problems in Teilprobleme und Lösen der Teilprobleme. Typische Teilprobleme sind etwa

- Eingabe: Einlesen der speziellen Aufgabe, z.B. vom Nutzer gewünschte Arbeitsschritte und zu bearbeitende Daten.
- Vorbearbeitung der Daten: Oft macht es Sinn, die Daten in eine bestimmte Repräsentationsform zu bringen oder einfache Sonderfälle abzuspalten bzw. außerhalb des eigentlichen Algorithmus zu behandeln. In manchen Fällen steckt hier die eigentliche Arbeit.
- Implementierung der zu verwendenden Algorithmen: Anwendung der ausgewählten oder konstruierten Algorithmen auf die Daten.
- Nachbearbeitung der Daten: Unter Umständen nützen die rohen Ergebnisdaten recht wenig; durch eine andere Repräsentation und weitere abgeleitete Größen werden sie besser handhabbar oder interpretierbar.
- Ausgabe des Ergebnisses: Von der Ausgabe über die Konsole über das Schreiben in eine Datei bis hin zu grafischer Ausgabe mit interaktiver Darstellung gibt es hier viele Möglichkeiten.

Diese Teilprobleme zerfallen in der Regel noch weiter, z.B. ist für das Addieren von Brüchen die Möglichkeit zum Kürzen der Brüche sinnvoll, für die man wiederum den größten gemeinsamen Teiler braucht. . .

Für das Zerlegen des Programmierprojektes in mehrere Dateien, die getrennt kompiliert und erst am Schluss zusammengebunden werden, sprechen vor diesem Hintergrund:

- Übersichtlichkeit: In sehr langen Dateien kann man den Überblick nur schlecht bewahren. Stehen die Lösungen für die Teilprobleme in eigenen Dateien, ist das leichter möglich.
- Wiederverwendbarkeit: Elementare Teilprobleme wie Ein- und Ausgabe treten immer wieder auf. Sind die entsprechenden Funktionen in einer eigenen Datei aufbewahrt, kann man sie leicht auch anderen Programmen zugänglich machen.
- Wartbarkeit: Werden an Stelle lokaler Kopien oft verwendeter Funktionen diese Funktionen zentral in einer Datei vorgehalten, kommen Korrekturen und Ergänzungen allen Programmen gleichzeitig zugute und es bestehen weniger Möglichkeiten, Verbesserungen zu vergessen.

| | |
|---|---|
| *Ausgehend von `.c`/`.i`-Dateien* | |
| `-E` | Nur Präprozessor-Lauf |
| `-S` | `-E` plus Kompilierung |
| `-c` | `-S` plus Assemblierung |
| `-o` *Name* | Name der Zieldatei |
| Standard: `-c` plus Linken, Zieldatei `a.out` | |
| `-x Sprache` | Programmiersprache (für andere Endungen als `.c`/`.i`) |
| `-std=`*Standard* | Welcher ***C***-Sprachstandard soll eingehalten werden?<br>**`c89`** Ansi-***C***, wie 1989 definiert<br>**`iso9899:199409`** Ansi-***C*** plus Amendment 1<br>**`gnu89`** ***C***89 plus gnu-Erweiterungen (Standard-Einstellung)<br>**`c99`** Ansi-***C***, wie 1999 definiert; noch nicht vollständig implementiert (siehe auch `http://gcc.gnu.org/c99status.html`)<br>**`gnu99`** ***C***99 plus gnu-Erweiterungen (wird Standard, wenn ***C***99 fertig implementiert) |
| *Präprozessor-Optionen* | |
| `-C` | Kommentare nicht entfernen (mit `-E`) |
| `-D`*macro* | Definiert `macro` wie eine `#define macro`-Anweisung |
| `-D`*macro=defn* | dito, für `#define macro defn` |
| `-U`*macro* | Bewirkt `#undef macro` |
| *Warnungen* | |
| `-fsyntax-only` | Nur Syntax-Fehler anmahnen |
| `-pedantic` | Stärkerer Test auf ISO-Kompatibilität |
| `-Wformat` | `printf`- und `scanf`-Format-Konsistenzprüfung |
| `-Wall` | Obiges plus unbenutzte Objekte plus sonstige Standardfehler |
| *Optimierung* | |
| `-O`, `-O1` | Optimierungen (bessere Ausnutzung der Prozessor-Register etc) |
| `-O2` | Optimierungen ohne Geschwindigkeitsgewinn auf Speicherkosten (sowie ohne *loop unrolling* und *function inlining*) |
| `-O3` | Schaltet auch die Optimierungsoption `-finline-functions` an |
| *Debugging und Profiling* | |
| `-g`, `-g`*Level* | Debug-Informationen im Code belassen; verträgt sich im Unterschied zu anderen Compilern mit Optimierung, Standard-Wert für Ausführlichkeits-*Level* ist 2 |
| `-ggdb`, `-ggdb`*Level* | dito, mit mehr Informationen für den Debugger *gdb* |
| `-p` | Informationen für den Profiler *prof*, so dass herausgefunden kann, welche Teile des Programms wieviel Zeit kosten |
| `-pg` | dito, für *gprof* |
| *Pfade* | |
| `-I`*Pfad* | Zusätzlicher Suchpfad für Include-Dateien |
| `-L`*Pfad* | dito für Bibliotheken, die hinzugelinkt werden sollen |
| `-l`*Bibliothek* | Linke die Bibliothek hinzu. Unter Linux ist der Bibliotheksname `lib`*Bibliothek*`.a` |

**Tabelle 3.1.:** Einige Optionen des Compilers *gcc*.

- Klar definierte Schnittstellen: Durch die Trennung der einzelnen Funktionen(sammlungen) ist klar und sichergestellt, welche Daten wann in welcher Weise wohin „fließen“. Damit einher gehen die bessere Isolierbarkeit von Fehlern und das Sicherstellen, dass nur in bestimmter Weise auf bestimmte Daten und Funktionen zugegriffen wird. Zu ersterem: Sind Daten vor dem Aufruf einer Funktion in einer anderen Datei korrekt und danach nicht mehr, dann ist der Fehler schon auf diese Datei (und alle Dateien, auf deren Funktionen von dort aus zugegriffen wird) begrenzt. Zu letzterem: Wenn man von außen nur über bestimmte Funktionen in eine Datei gelangen kann, reicht es, Sicherheitsabfragen, die auf fehlerhafte Eingangsdaten testen, auf diese Funktionen zu beschränken, so dass Funktionen, die nur von innerhalb der Datei aufgerufen werden können und unter Umständen zeitkritisch sind, ohne diese Abfragen auskommen können bzw. diese Abfragen nur in einem Test- oder Debug-Modus per bedingter Kompilierung eingebunden werden.

- Ersetzbarkeit von Teillösungen: Gibt es eine klar definierte Schnittstelle (diese Daten gehen rein, diese Daten kommen bei korrekter Lösung raus), dann kann an dieser Schnittstelle auch eine andere Funktion mit den gleichen Daten aufgerufen werden, die z.B. einem anderen Algorithmus folgt.

- Aufteilbarkeit auf mehrere Programmierer: Mit klaren Schnittstellen und klar definierten Eingangs- und Rückgabedaten können mehrere Programmierer zusammen an der Lösung des Gesamtproblems arbeiten.

- Weitergebbarkeit: Man kann eigenen Code an andere Leute weitergeben bzw. Code von anderen Leuten verwenden, ohne dass größere Änderungen an vorhandenem Code notwendig werden.

Ein Modul besteht nicht notwendigerweise nur aus ein paar Quellcode-Dateien. Beispielsweise könnte ein Modul für die Verwaltung komplexer geometrischer Objekte zerfallen in eine Header-Datei, die die notwendigen Strukturen enthält sowie einfache Zugriffsmakros, in eine Datei, in der Werkzeuge zur Initialisierung und Manipulation bereitgestellt werden, nebst zugehörigem Header, eine Datei, die Testroutinen enthält, die bestimmte Eigenschaften (z.B. Konvexität, Sternförmigkeit, richtige Orientierung der Oberfläche) überprüfen, eine, die die Zuordnung von Freiheitsgraden zur Geometrie regelt, eine weitere, die darauf aufbauend für konkrete Probleme Vektoren und Matrizen assembliert, eine, die die Projektion auf ein zweidimensionales Objekt bezüglich einer bestimmten Betrachterposition und -blickrichtung durchführt und diese Ausgabe an ein Visualierungsmodul weiterleitet, usw.

Es ist äußerst sinnvoll, sich vor dem Programmieren zu vergegenwärtigen, was für Aufgaben anfallen, wie man sie in Teilaufgaben (Module) und einzelne Funktionen zerlegen kann. Dadurch kann der Programmieraufwand unter Umständen erheblich gesenkt werden. Mehr dazu findet sich in Abschnitt 9.2

**Beispiele** Die Datei, die das Hauptprogramm enthält, muss nicht besonders lang sein – es ist vielmehr nützlich, wenn die Programmstruktur klar wird. Beispielsweise könnte das Hauptprogramm lediglich Initialisierungsroutinen für verschiedene Module aufrufen und dann in ein Modul springen, das in eine interaktive Kommunikation mit dem Benutzer tritt und die vom Benutzer eingegebenen Befehle mit Hilfe der anderen Module ausführt.

- Das Programm `haupt` besteht aus `haupt.c` und dem Modul `meinalgo`, seinerseits bestehend aus `meinalgo.c` und `meinalgo.h`, und bindet die Header-Datei `meinalgo.h` ein, um auf die in `meinalgo.c` definierten Funktionen zugreifen zu können:
  ```
  gcc -Wall -std=c99 -o haupt haupt.c meinalgo.c
  ```
  Ändert sich etwa an `meinalgo.c` nichts mehr, so kann die *Turn-around*-Zeit für die Erstellung von `haupt` gedrückt werden, indem man `meinalgo.c` schon fertig kompiliert vorhält:
  ```
  gcc -Wall -std=c99 -c meinalgo.c
  ```
  Dann sieht die Erstellung von `haupt` folgendermaßen aus:
  ```
  gcc -Wall -std=c99 -o haupt haupt.c meinalgo.o
  ```

- Benötigt `meinalgo` die Bibliothek `libmeinebib.a`, so muss diese abschließend hinzugelinkt werden:
  ```
  mygcc -o haupt haupt.c meinalgo.o -lmeinebib
  ```
  Hierbei ist `mygcc` ein Alias für `gcc -Wall -std=c99`. Ist für `libmeinebib.a` wiederum die Mathe-Bibliothek `libm.a` notwendig, so muss diese ihrerseits *nach* `libmeinebib.a` gelinkt werden, da der Linker von links nach rechts arbeitet:
  ```
  mygcc -o haupt haupt.c meinalgo.o -lmeinebib -lm
  ```
  Der Aufruf
  ```
  mygcc -o haupt haupt.c meinalgo.o -lm -lmeinebib
  ```
  dagegen führt dazu, dass alle Symbole (z.B. Funktionsnamen), die in `haupt.o` und `meinalgo.o` nach dem Kompilieren noch unbekannt sind und aus der `libm.a` stammen (etwa `sqrt()`), aufgelöst werden, und anschließend alle Symbole, die in `haupt.o`, `meinalgo.o` `libm.a` noch nicht definiert sind, mit Hilfe der `libmeinebib.a` aufgelöst werden. Abschließend wird immer noch die Standardbibliothek hinzugelinkt. Sollte nun in `libmeinebib.a` ein Symbol aus `libm.a` vorkommen, das noch nicht aufgelöst ist, dann stellt der Linker am Ende fest, dass dieses Symbol fehlt.
  Unsauber programmierte Bibliotheken können sich wechselseitig bedingen, dann werden Konstruktionen wie `-lfoo -lbar -lfoo` notwendig.

Wenn viele Module verwendet werden, wird der Aufruf zum Erstellen der ausführbaren Datei recht hoch. Hierfür verwenden komplexe Projekte so genannten *Makefiles*, die Abhängigkeiten und eine „Bauanleitung“ enthalten. Siehe dazu auch Abschnitt 8.2.

## Beispielprogramme

Beispielprogramm in mehreren Dateien.

***Programmlisting 3.1:*** radiuspruef.h

```
/*************************************************************************
**
** radiuspruef - Modul fuer Radius-Beschraenkung
**
*************************************************************************/

/* Safe include */
#ifndef __RADIUSPRUEF_H__
    // __RADIUSPRUEF_H__ noch nicht definiert, d.h. die Datei
    // wurde noch nicht eingebunden.
#define __RADIUSPRUEF_H__
    // __RADIUSPRUEF_H__ definieren, damit die Datei nicht ein
    // zweites Mal eingebunden wird und die Konstanten bzw. Makros
    // doppelt definiert werden.

/*** INCLUDES & DEFINES ****************************************/

#define R_MIN 1.0
#define R_MAX 20.0

/*** Funktionsprototypen ***************************************/

int radiuspruef (double radius);

#endif
```

***Programmlisting 3.2:*** radiuspruef.c

```
/*************************************************************************
**
** radiuspruef - Modul fuer Radius-Beschraenkung
**
*************************************************************************/


#include <stdio.h>
#include "radiuspruef.h"        // eigene Header-Datei einbinden

/*************************************************************************
**
** Funktion radiuspruef - prueft, ob der Radius innerhalb der
** vorgegebenen Grenzen liegt
**
** Argumente: radius .. Radius der Kugel in doppelter Genauigkeit
**
** Rueckgabe: 0 .. Radius innerhalb der geforderten Grenzen
**            1 .. Radius zu klein/gross bzw. negativ
**
*************************************************************************/
```

```
int radiuspruef (double radius)
{
    if (radius < 0.0) {
        printf("Radius darf nicht negativ sein!\n");
        return(1);
    }
    else if (radius < R_MIN) {
        printf("Radius ist zu klein!\n");
        return(1);
    }
    else if (radius > R_MAX) {
        printf("Radius ist zu gross!\n");
        return(1);
    }
    else
        return(0);
}
```

***Programmlisting 3.3:*** volumina.h

```
/*****************************************************************
**
** volumina - Berechnung von Kreis-/Kugelvolumen
**
*****************************************************************/

/* Safe include */
#ifndef __VOLUMINA_H__
#define __VOLUMINA_H__

/*** INCLUDES & DEFINES ******************************************/

#include <math.h>   // fuer M_PI; ist aber nicht im C-Standard drin!

#ifndef M_PI        // PI als double zur Verfuegung stellen
#define M_PI 3.14159265358979323846264338327950288
#endif

#define KREISFLAECHE(r) (M_PI * (r) * (r))
#define KUGELVOLUMEN(r) ((4.0 / 3.0) * M_PI * (r) * (r) * (r))

/* Debug-Modus an? (Auskommentieren falls nicht gewuenscht) */
#define PRUEFEN

#endif
```

***Programmlisting 3.4:*** volumina2.c

```
/*****************************************************************
**
** volumina2 -- Berechnung Kreis- und Kugelvolumen
**
** Kompilierung: mit gcc volumina2.c radiuspruef.c
**               bzw. gcc volumina2.c radiuspruef.c -DPRUEFEN
```

```
**
**               bzw. build volumina2 radiuspruef.c
**               bzw. build volumina2 "radiuspruef.c -DPRUEFEN"
**
** Geschichte:
**
**             2004-04-28: Nichtpermanenter Debug-Modus (M. Mair)
**             2004-04-19: Leben eingehaucht durch J. Werner
**
*****************************************************************/

/*** INCLUDES & DEFINES ******************************************/

#include <stdio.h>
#include "volumina.h"       // Makros fuer 2D-/3D-Volumenberechnung
#include "radiuspruef.h"        // radiuspruef(), Radius-Beschraenkung

/*** Funktionsprototypen *****************************************/

/*****************************************************************
**
** Hauptprogramm
**
** Argumente: Keine
**
** Rueckgabe: 0 .. kein Fehler
**
*****************************************************************/

int main (void)
{
    double radius, flaeche, volumen;

    printf("Beispielprogramm zur Berechnung von Kreisflaechen und");
    printf(" Kugelvolumina\n");

    printf("Bitte geben Sie fuer den Radius einen Wert ");
    printf("zwischen %.2f und %.2f an: ", R_MIN, R_MAX);

    scanf("%lf", &radius); // Eingabe einlesen, Wert -> radius

    /* folgender Codeblock wird nur ausgefuehrt, wenn die Konstante
    ** PRUEFEN in der Header-Datei "volumina.h" definiert wurde */
    //#ifdef PRUEFEN
    /* wiederhole solange bis der Wert von radius gueltig ist */
    printf("hier\n");
    while (radiuspruef(radius)) {
        printf("Bitte geben Sie fuer den Radius einen neuen Wert ");
        printf("zwischen %.2f und %.2f an: ", R_MIN, R_MAX);

        scanf("%lf", &radius); // Eingabe einlesen, Wert -> radius
    }
    //#endif

    printf("Berechne die Flaeche eines Kreises mit dem Radius");
    printf(" r = %f mm\n", radius);
```

```
    flaeche = KREISFLAECHE(radius);
    printf("Die Flaeche A betraegt %f mm2\n", flaeche);

    volumen = KUGELVOLUMEN(radius);
    printf("Das Volumen V betraegt %f mm3\n", volumen);

    return(0);
}
```

# 4. Datentypen

Die Größe eines ***C***-Datentyps wird in Byte gemessen. Der `sizeof`-Operator liefert diese Größe zurück. Das einzige, was man immer mit Sicherheit weiß, ist, dass `sizeof(char)` immer 1 liefert. Der Rückgabe-Typ von `sizeof` ist formal `size_t`, wobei `size_t` kein „echter“ ***C***-Datentyp ist, sondern nur (vor-)definiert ist, so dass *alle* möglichen Objektgrößen darin enthalten sein können (siehe Abschnitt 4.18). Der Standard gibt mehrere solcher Datentypen vor; man erkennt sie an dem `_t` am Ende. Sie werden mittels `typedef` definiert (siehe 4.17).

Konstanten zu den jeweiligen Datentypen werden gesammelt in Abschnitt 4.14 behandelt.

## 4.1. Leerer Datentyp

Der leere Datentyp `void` zeigt in Funktionsprototypen und -definitionen, dass der Rückgabetyp leer ist, die Funktion also nichts zurückgibt, oder dass die Parameterliste leer ist, es also keine Parameter gibt. Zusätzlich wird er in Verbindung mit Zeigern verwendet, um auf ein nicht vorher festgelegtes Objekt zu zeigen, siehe Abschnitt 4.7 und 4.16.

## 4.2. Zeichen

Für Zeichen wird standardmäßig der Byte-Datentyp `char` verwendet. Aufgrund der Beschränkung durch die Byte-Größe, die rechnerabhängig und in der Regel zu gering als Container für alle Zeichen aller Sprachen ist, liegt nur ein geringer Teil des Zeichenumfangs fest. Für ***C*** gibt es auch eine Umschrift für Zeichen, die nicht notwendigerweise im vorgegebenen Umfang enthalten sind, aber zum Sprachumfang gehören.

Zur Lösung der Probleme bei der Stringbehandlung wurden sogenannte Multibyte-Zeichen eingeführt, die durch `char`s dargestellt werden. Oft dient hierbei ein Teil der Bits des ersten Bytes als Zeichen dafür, dass ein zweites Byte vorhanden ist. Zudem gibt es auch Zeichensätze, die eine feste Bitanzahl verwenden und auf allen Rechner gleich sind (aber nicht unbedingt gleich implementiert).

Für den Umgang mit Multibyte-Characters und -Strings wurde der „weite“ Zeichen-Datentyp `wchar_t` (*wide character*) eingeführt, der die eigentlichen Zeichen und Zeichengrößen verbirgt und einen Umgang mit Multibyte-Zeichensätzen gleich wie für einen gewöhnlichen Ein-Byte-Zeichensatz ermöglicht. Dieser Datentyp ist implementierungsabhängig, für die Ein- und Ausgabe werden `wchar_t`-Zeichen in Multibyte-Zeichen umgewandelt bzw. umgekehrt. Aus Bequem-

lichkeitsgründen ist der Stringterminator der gleiche wie bei normalen Strings (also `\0`)[1].

Der ***C***-Standard garantiert, dass ein Byte mindestens 8 Bits hat. Die genaue Anzahl steht in der symbolischen Konstante `CHAR_BIT`, die im Header `<limits.h>` definiert wird.

## 4.3. Ganzzahl-Datentypen

Der generische Ganzzahl-Datentyp eines Rechners ist `int` und ist normalerweise der größtmögliche Ganzzahl-Datentyp, mit dem der Prozessor intern rechnen kann. Ein `int` besteht aus einem oder mehreren Bytes[2].

In ***C*** sind die Ganzzahl-Datentypen `char`, `short int`, `int` und `long int` bekannt. Statt `short int` und `long int` kann man auch `short` bzw. `long` schreiben. Die Größe in Byte hängt bei `short` und `long int` wieder vom Rechner und vom Compiler ab[3], wobei `short` mindestens so groß ist wie `char`, `int` wie `short` und `long` wie `int`. Durch das Voranstellen von `unsigned` (oder `signed`) wird angegeben, dass es sich um einen vorzeichenlosen (vorzeichenbehafteten) Ganzzahl-Datentyp handelt. Ein 16-Bit-**`short`** nimmt beispielsweise Werte von $-2^{15}$ bis $+2^{15}-1$ an, das zugehörige `unsigned short` Werte von 0 bis $2^{16}-1$. Statt `unsigned int` und `signed int` kann man auch nur `unsigned` bzw. `signed` schreiben.

Mit ***C***99 kommt noch der Typ `long long int` (bzw. `long long`) hinzu, der mindestens so groß ist wie `long`. Für bestimmte Anwendungen ist eine garantierte Integer-Größe wichtig; hierfür stehen bei Einbinden von `<stdint.h>` neben dem größten Integer-Datentyp `intmax_t` und seinem vorzeichenlosen Pendant `uintmax_t` die Integer-Typen `int`*N*`_t`, `uint`*N*`_t`, `int_least`*N*`_t`, `uint_least`*N*`_t`, `int_fast`*N*`_t` und `uint_fast`*N*`_t` zur Verfügung, wobei für *N* auf jeden Fall 8, 16, 32 und 64 eingesetzt werden kann.

Leider werden die „nützlichsten“ Typen mit fester Bitzahl nicht vom Standard garantiert. Um zu testen, ob man den gewünschten Typ hat, kann man folgendermaßen vorgehen: Man schaut nach, ob das Minimum oder Maximum definiert ist:

```
#ifdef __STDC_VERSION__ && __STDC_VERSION__ >= 199901L
# include <stdint.h>
# ifdef INT32_MAX
#  define myint32 int32_t
# else
#  define myint32 int_least32_t
# endif
#else
# define myint32 long
#endif
```

[1] Nähere Informationen stehen in der entsprechenden man-page. Falls diese auf dem Rechner nicht vorhanden ist, einfach im Internet nach `man multibyte` suchen.

[2] Aktuelle 32- oder 64-Bit-Architekturen mit einer Byte-Größe von 8 Bit haben also eine **`int`**-Größe von 4 bzw. 8 Byte.

[3] Typische Größen sind etwa bei 32-Bit-Architekturen 16-Bit-`short`s und 32-Bit-`long`s.

Normalerweise verwendet man allerdings statt **`#define`** eine Typdefinition über **`typedef`**, siehe Abschnitt 4.17.

## 4.4. Wahrheitswerte

In Standard-***C*** werden Ganzzahl-Datentypen als Wahrheitswerte interpretiert, wobei Null für falsch steht und alle anderen Werte für wahr. Mit ***C***99 gibt es jetzt auch boolesche Variablen, die nur die Werte 1 und 0 kennen, der entsprechende Ganzzahl-Datentyp ist `_Bool`. In anderen Programmiersprachen gibt es echte Wahrheitswerte *true* und *false*, das ist in ***C*** nicht der Fall. Fängt man neu mit einem Programm an, sollte man `<stdbool.h>` einbinden. Dieser Header definiert die Makros `bool`, `true` und `false`.

## 4.5. Fließkommazahlen

Fließkommazahl-Datentypen unter ***C*** sind `float` für einfach genaue und `double` und `long double` für doppelt genaue Fließkommazahlen. Die genauen Zahlenbereiche sind implementierungsabhängig, aber man hat garantiert Zahlen von $-10^{37}$ bis $-10^{-37}$ sowie von $10^{-37}$ bis $10^{37}$ und die 0 mit einer Genauigkeit von sechs bzw. zehn Nachkommastellen zur Verfügung. In der Regel sind `float`s 32 Bit groß und `double`s 64 Bit, da es einen IEEE-Standard[4] für einfach und doppelt genaue Maschinenzahlen gibt, der von diesen Größen ausgeht. Typische Größen für `long double`s sind 80 oder 128 Bit. Fließkommazahlen werden standardmäßig als `double`s aufgefasst.

*Warnung:* Fließkommazahlen im Rechner sind diskrete Zahlenwerte aus einem endlichen, rationalen Wertebereich und sollten nicht mit rationalen oder gar reellen Zahlen verwechselt werden! Schon die Repräsentation einer Zahl durch eine Fließkommazahl ist nicht exakt – der Wert 0.1 beispielsweise kann selbst im `long double`-Zahlenbereich nicht exakt dargestellt werden. Vergleiche auf Gleichheit sollten immer nur in Verbindung mit einer anwendungsspezifischen Genauigkeit durchgeführt werden, d.h. zwei Variablen `a`, `b` werden etwa als gleich angesehen, wenn der Betrag der Differenz einen geringen relativen Fehler aufweist:

```
#include <stdbool.h>

#define PRECISION_F (1e-5)
#define PRECISION_D (1e-10)
#define PRECISION_L (1e-15)

bool IstGleich_f (float a, float b)
{
    if (fabsf(a-b)<=PRECISION_F*(fabsf(a)+fabsf(b))
        return true;
    else
        return false;
```

[4] Einen guten Überblick findet man unter *http://www.cs.berkeley.edu/~wkahan/ieee754status/*.

```
}

bool IstGleich_d (double a, double b)
{
    if (fabs(a-b)<=PRECISION_D*(fabs(a)+fabs(b))
        return true;
    else
        return false;
}

....
    double obereSchranke, untereSchranke;

    ....
    if (IstGleich_d(obereSchranke, untereSchranke) {
        ....
    }
    ....
```

## 4.6. Komplexe Zahlen

Mit ***C***99 gibt es auch komplexe Fließkommazahl-Datentypen: `float _Complex`, `float _Imaginary`, `double _Complex`, `double _Imaginary`, `long double _Complex`, `long double _Imaginary`. Die Schlüsselwörter sind mit führendem Unterstrich versehen und am Anfang groß und dann klein geschrieben, weil viele bestehende ***C***-Programme bereits eigene komplexe Datentypen verwenden, die beispielsweise Namen von `complex` bis `__COMPLEX__` haben könnten.

Wird `<complex.h>` eingebunden, stehen die Schlüsselwörter `complex` und `imaginary` zur Verfügung. Das empfiehlt sich für alle neuen Programme.

## 4.7. Zeiger und Referenzen

Ein Zeiger „zeigt“ auf eine Adresse im Speicher des Rechners. In ***C*** ist dem Zeiger dabei auch ein Typ zugeordnet, so dass klar ist, als was für ein Datentyp die Daten an der entsprechenden Adresse interpretiert werden sollen und damit auch, wie groß das entsprechende Objekt ist.

### 4.7.1. Referenzierung / Dereferenzierung

Ein Zeiger wird deklariert durch Hinzunahme des Sterns `*`:

```
int *iptr;
float gkz, *fptr;
struct MyStruct *MSptr;
```

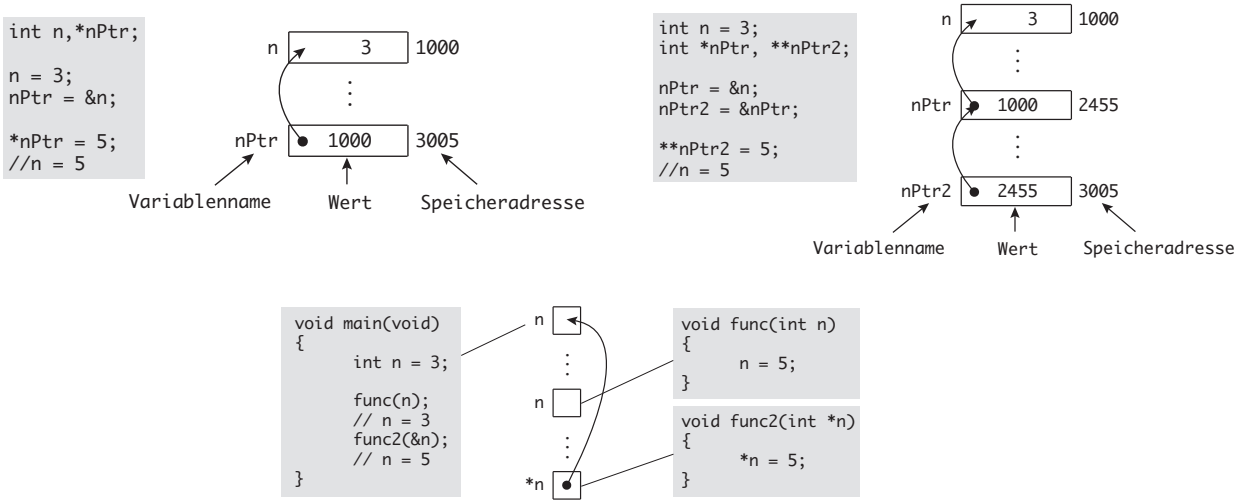



**Abbildung 4.1.:** Oben: Zeiger und Wert. Unten: Sinn von Referenzen.

```
gkz  = 5.0;
fptr = &gkz;
printf("Wert von fptr: %f\n", *fptr);
```

Außerhalb der Deklaration ist `*` der sogenannte Dereferenzierungs- oder Inhaltsoperator; er liefert den Wert, der an der Speicheradresse steht, auf die der Zeiger zeigt und interpretiert ihn als entsprechenden Typ. Das Gegenstück ist der Referenzierungs- oder Adressoperator &, der die Adresse einer Variable liefert. Konstanten dürfen nicht referenziert werden. Abbildung 4.1 verdeutlicht noch einmal den Zusammenhang von Zeiger, Wert und Referenz.

### 4.7.2. Zeigerarithmetik

Die Deklaration des Zeigers führt nur zur Bereitstellung des Speicherplatzes, der den Zeiger enthält, nicht aber zur Bereitstellung des zusätzlichen Speicherplatzes, auf den der Zeiger zeigt. Zuweisungen a la

```
int *iptr;

*iptr = 5;
```

führen dazu, dass irgendeine Stelle im Speicher mit dem `int`-Wert 5 überschrieben wird. Welche, hängt davon ab, welcher Wert an der Stelle `iptr` bei der Deklaration stand. Ein beliebter anderer Fehler ist das Verwenden eines Zeigers, um ein Array oder andere Datenstrukturen zu durchlaufen, in Verbindung mit dem über das Ende bzw. über den Anfang Hinausschreiben. Das kann bei Verwendung von Indizes zwar auch passieren, aber mit Zeigern kann man es besser verbergen…

```
int *iptr, arr[20];

....
```

```
arr[19] = 0;    // Stringterminator ans Ende

// Der zweite Zeiger schreibt an die Adresse von
// arr[20]
for (iptr=&arr[1]; *iptr>0; iptr++)
    arr[0] += function(iptr,iptr+1);
```

Die Anweisung

```
iptr++
```

ist wie üblich äquivalent zu

```
iptr = iptr + 1
```

und wird als

```
iptr = (int *) ( ((char *)iptr) + sizeof(int) )
```

interpretiert, d.h. die Adresse, auf die iptr zeigt, erhöht sich um `sizeof(int)`; das läuft für andere Zeigertypen natürlich analog[5]. Der Grund ist, dass für `sizeof(int)` $> 1$, die Adresszelle direkt nach dem ersten Byte der Variable auf das zweite Byte der Variable zeigt usw., d.h. ein logisch sinnvolles Weiterschalten der Adresse um 1 muss um so viele Bytes erfolgen, dass nicht mehr auf Bytes des ursprünglichen Objektes gezeigt wird. Das ist bei Feldern ja genau das gleiche – eine Erhöhung des Feldindexes führt zum nächsten Feldelement. Daher funktionieren Array-Zugriff und -Indizierung auch für Zeiger; die Anweisungszeile von oben könnte also auch

```
arr[0] += function(iptr,&iptr[1]);
```

lauten, mit `iptr[0]` erhält man den Wert der Variable, auf die der Zeiger zeigt, das heißt `iptr[`*index*`]` ist das gleiche wie `iptr`$+$*index*.

Fehler mit nicht initialisierten Zeigern kann man umgehen, indem man einem Zeiger zunächst einmal den Wert `NULL` zuweist. Diese Konstante ist meistens als `(void *) 0` oder manchmal einfach nur `0` definiert. Dieser Zeiger-Wert bedeutet, dass der Zeiger momentan auf nichts zeigt und wird auch als Rückgabewert von Funktionen verwendet[6].

Wenn das Betriebssystem dafür sorgt, dass ein Programm nur in einem bestimmten Speicherbereich arbeiten darf, können Fehler mit nicht initialisierten Zeigern leichter gefunden werden. Fehler der zweiten Art sind dagegen oft nicht so leicht zu finden; man überschreibt sich unter Umständen einfach eigene Variablen oder gar Programmteile.

[5] Diese spezielle Rechenweise sowie das Rechnen mit Zeigern allgemein findet sich in Büchern oft unter dem Begriff Zeiger-Arithmetik.

[6] Das bedeutet übrigens nicht notwendigerweise, dass es sich hierbei um die physikalische Adresse 0 des Speichers handeln muss – `NULL`/`0` ist zur Programmlaufzeit durch eine Kombination von Bits repräsentiert, die von Hardware und Betriebssystem abhängt.

### 4.7.3. Felder, Teil 2

Felder (*Arrays*) wurden bereits in Abschnitt 2.3.4 eingeführt und die designierte Initialisierung in Abschnitt 4.12, deswegen hier nur noch einmal in Kürze: Felder können über beliebigen Datentypen erzeugt werden, sie werden mittels *Datentyp feldname*[*feldgröße*] deklariert. Auf einzelne Elemente des Feldes kann durch Indizierung des Feldnamens zugegriffen werden:

```
int test, feld[20];

....

test = feld[0];
```

Der Feldindex startet bei 0 und läuft bis *feldgröße*−1. In ***C*** 89 können Felder nicht dynamisch alloziert werden, d.h. die Feldgröße steht beim Kompilieren fest; in ***C***99 sind dynamische Felder möglich.

Felder können nur bei der Deklaration als ganzes definiert werden, i.d.R werden hierbei die Feldelemente in geschweiften Klammern durch Kommata getrennt angegeben,

```
int feld[] = {1,2,3};
```

wobei dann die Feldgröße in der Deklaration weggelassen werden kann. Später kann nur noch elementweise zugewiesen werden.

Eine Ausnahme bilden ***C***-Strings; das sind `char`-Arrays, innerhalb derer eine Zeichenkette durch den Stringterminator, `'\0'` definiert ist. Alternativ zu

```
char string[] = {'s','t','r','i','n','g','\0'};
```

können sie auch durch Anführungszeichen als String gekennzeichnet werden:

```
char string[] = "string";
```

Die Länge beider Strings ist identisch, nämlich in diesem Beispiel 6; man benötigt zusätzlich noch Platz für den Stringterminator, d.h. das Feld muss mindestens 7 Byte umfassen. Für das Arbeiten mit Strings gibt es einige Funktionen in der Standard-Bibliothek, siehe auch Abschnitt 2.3.4.

Der Feldname selbst wird, wenn er auf der rechten Seite einer Zuweisung steht, bzw. als Argument an eine Funktion übergeben wird, als Referenz des ersten Feldelements betrachtet, d.h. er steht für die Adresse des ersten Feldelements. Siehe hierzu auch den folgenden Abschnitt.

In ***C***99 sind auch dynamische Arrays möglich; darüber hinaus darf das *letzte* Element einer Struktur ein *flexibles Array-Element*, eine Art dynamisches Array, sein; hierbei muss dann der Speicher für die Struktur plus das Array alloziert werden:

```
struct Beispiel {
    double v;
    int w;
    short flex[];
};
```

```
struct Beispiel *Bzeiger = NULL;
size_t flexgroesse = 0;

....

Bzeiger = malloc( sizeof(struct Beispiel)
                  + flexgroesse*sizeof(short) );
```

Diese Erweiterung entspricht aber nicht der gängigen Praxis von Compilererweiterungen, weswegen sie z.B. von *gcc* noch nicht unterstützt wird.

Felder als Funktionsparameter werden in Abschnitt 6.7 besprochen. Hier gibt es interessante ***C***99-Neuerungen.

### 4.7.4. „Call by value“ und „call by reference“

Für Funktionen gibt es in ***C*** nur einen möglichen Rückgabewert. Soll mehreren Variablen durch einen Funkionsaufruf ein Wert zugewiesen werden, müsste man entweder alle global bekannt machen oder eine Struktur zurückgeben, aus der dann jeweils die initialisierten Werte ausgelesen werden. Das hat gravierende Nachteile, was die Übersichtlichkeit und „Sicherheit“ des Programms angeht.

Indem man nun nicht den Wert einer Variablen („call by value“) übergibt, sondern die Speicheradresse der Variablen („call by reference“), kann die Funktion den neuen Wert an diese Adresse schreiben und nach dem Ende der Funktion steht in der Variablen der richtige Wert. Anders gesagt: Wenn eine Variable durch eine Funktion *geändert* werden soll, muss man der Funktion eine Möglichkeit geben, die Stelle im Speicher zu ändern, an der die Variable steht! Die Variablen aus der Argumentliste einer Funktion enthalten nur Kopien der Werte, die beim Aufruf der Funktion übergeben werden. Indem man die Adresse einer Variablen übergibt, entsteht zwar wieder nur eine Kopie der Adresse, aber das reicht aus, um den Wert, der an dieser Adresse steht, zu ändern. Ein Beispiel für diesen Vorgang ist die `scanf`-Funktion, der man Referenzen auf die Variablen, in die der eingelesene Wert hineingeschrieben wurde, übergibt.

Im folgenden Programmschnipsel wird ein Feld, das *n* zweidimensionale Koordinaten enthalten kann, mit Punkten gefüllt, die auf dem Rand einer Ellipse liegen.

```
#define DIM (2)

int ellipse_init(int n, double *coords, double dir1[DIM],
                 double dir2[DIM], double mp[DIM])
{
    int i, j;

    for (i=0; i<n; i++)
        for (j=0; j<DIM; j++)
            coords[i*DIM+j] = mp[j] + dir1[j]*cos(2*PI*i/n)
                                    + dir2[j]*sin(2*PI*i/n);
```

```
    return(0);
}

int main (void)
{
    double richt1[DIM], richt2[DIM], Koord[NUMPOINTS*DIM];

    ....
    if(ellipse_init(NUMPOINTS, Koord, richt1, richt2))
        printf("FEHLER\n");
    ....
}
```

Man kann statt `double *coords` auch `double coords[]` schreiben – die Bedeutung ist die selbe. Dagegen ist `double coords[n]` nicht möglich, weil *n* zur Laufzeit festgelegt wird. Mit Konstanten dagegen funktioniert das; auf diese Weise lassen sich etwa die Koordinaten als $n \times DIM$-Feld übergeben, wenn einem das lieber ist:

```
#define DIM (2)

int ellipse_init(int n, double *coords[DIM], double dir1[DIM],
                 double dir2[DIM], double mp[DIM])
{
    int i, j;

    for (int i=0; i<n; i++)
        for (int j=0; j<DIM; j++)
            coords[i][j] = mp[j] + dir1[j]*cos(2*PI*i/n)
                                 + dir2[j]*sin(2*PI*i/n);
    return(0);
}

int main (void)
{
    double richt1[DIM], richt2[DIM], Koord[NUMPOINTS][DIM];

    ....
    if(ellipse_init(NUMPOINTS, Koord, richt1, richt2))
        printf("FEHLER\n");
    ....
}
```

Der Typ von `double *coords[DIM]` ist „Zeiger auf (ein) `double`-Feld mit *DIM* Elementen“ und könnte in der Funktionsdefinition auch als `double coords[][DIM]` geschrieben werden. Wäre *DIM* nicht konstant, dann wäre das nicht möglich – man wüsste nicht, wie gross das Objekt ist, auf das der Zeiger zeigt.

In C können keine Arrays übergeben werden, sondern nur Zeiger auf das erste Feldelement.[7] Ausserdem muss die Größe eines Arrays bekannt sein. Mehrdimensionale Arrays lassen sich nicht mit flexibler Größe direkt über Zeiger übergeben, da sie keine direkte Entsprechung als Zeiger haben: Jeder Eintrag eines Arrays enthält einen Wert, während ein Zeiger auf einen Zeiger sozusagen zwei Speicherplätze für Zeiger und einen für den Wert braucht (siehe auch Abbildung 4.1)[8].

### 4.7.5. Dynamische Speicher-Allokation

Manchmal will man Speicher erst während des Programmlaufs allozieren, weil man auf Nutzereingaben reagieren muss oder nur zeitweilig bestimmte Variablen/Felder braucht. Einen solchen Speicherplatz kann man mit den Funktionen `malloc()` bzw. `calloc()` anfordern und muss ihn später wieder mit `free()` freigeben.

```
#include <stdlib.h>    // fuer exit() und EXIT_FAILURE
                       // sowie malloc() und free()

#define STANDARD_GROESSE  20

int main (void)
{
    double *vptr, **mptr;
    size_t i, vsize, msize;

    printf("Wie gross sollen die Vektoren sein? ");
    if (scanf("%d",&vsize)!=1)
        vsize = STANDARD_GROESSE;
    if( (vptr = malloc(vsize*sizeof(double))) == NULL ) {
        printf("Kann keinen Speicher fuer Vektoren")
        printf(" allozieren");
        exit(EXIT_FAILURE); // Programm abbrechen
    }
    printf("Wieviele Matrix-Zeilen? ");
    if (scanf("%d",&msize)!=1)
        msize = STANDARD_GROESSE;
    if( (mptr = malloc(msize*sizeof(double *))) == NULL) {
        printf("Kann keinen Speicher fuer Matrixzeilen");
        printf(" allozieren");
        free(vptr); // Speicher freigeben
        exit(EXIT_FAILURE);
    }
    for (i=0; i<msize; i++) {
        if( (mptr[i] =
                  malloc(vsize*sizeof(double))) == NULL) {
```

[7] Eine Ausnahme bilden in Strukturen verpackte Felder – das ist aber unpraktisch, da man statt nur einer Adresse alle Elemente zwischenspeichern muss.

[8] Unter *C*99 gibt es eine Möglichkeit zur Übergabe flexibler Feldgrößen, siehe Abschnitt 6.7.

```
                printf("Kann keinen Speicher fuer");
                printf(" %d-te Matrixzeile allozieren",
                       i);
                for (size_t j=0; j<i; j++)
                    free(mptr[j]);
                free(mptr);
                free(vptr);
                exit(EXIT_FAILURE);
            }
        }

        ....
        for (i=0; i<msize; i++)
            free(mptr[i]);
        free(mptr);
        free(vptr);

        exit(EXIT_SUCCESS); // Fuer unsere Zwecke aequivalent
                            // zu  return 0;
    }
```

Die Funktion `malloc()` hat als Rückgabetyp `void *`. Dieser Zeigertyp kann in jeden beliebigen anderen Zeigertyp umgewandelt werden und umgekehrt – weil er aber nur eine Adresse beschreibt und nicht auch noch den zugehörigen Typ, ist kein Rechnen (Zeiger-Arithmetik) mit ihm möglich. Will man mit dem Zeiger rechnen, sollte man `char *` bzw. `unsigned char *` verwenden (andere Zeigertypen sind ebenfalls nicht sicher, siehe Abschnitt 4.18). Wenn man die Darstellung eines Objekts anschauen will, kann das etwa so aussehen:

```
unsigned char *byte;

byte = &objekt;
printf("Speicherdarstellung von objekt:\n");
for (size_t i=0; i<sizeof objekt; i++)
    printf("\t%x\n", byte++);
```

`calloc()` alloziert einen Speicherbereich und initialisiert ihn mit Nullen. Argumente sind hier Anzahl der Objekte und Größe der Objekte.

`realloc()` kann die Größe eines allozierten Speicherbereichs ändern. Argumente sind der alte Zeiger auf den Speicherbereich und die neue Größe, zurückgegeben wird ein Zeiger auf den neuen Speicherbereich. Bis zum Minimum aus alter und neuer Größe bleibt der Inhalt gleich. Hier muss man aufpassen, falls mehr als ein Zeiger auf den alten Speicherbereich gezeigt hat.

Es gibt übrigens eine geschickte Möglichkeit, den `sizeof`-Operator für eine wartungsfreundlichere Allozierung zu verwenden: Man kann `sizeof` auch auf eine Variable anwenden, um ihren Wert herauszubekommen; wendet man `sizeof` auf den Inhalt des Zeigers an, der auf den allozierten Speicherbereich zeigen soll, so ist der Typ, auf den der Zeiger zeigt, indirekt beschrieben – und muss, falls er sich ändert, nicht im `malloc()`-Aufruf geändert werden:

```
#include <stdlib.h>
....
int main ()
{
    double *vptr, **mptr;
    size_t i, vsize, msize;

    ....
    if( (vptr = malloc(vsize*sizeof *vptr)) == NULL ) {
        ....
    }
    ....
    if( (mptr = malloc(msize*sizeof *mptr)) == NULL) {
        ....
    }
    for (i=0; i<msize; i++) {
        if( (mptr[i] =
                  malloc(vsize*sizeof **mptr)) == NULL) {
            ....
        }
    }

    ....
}
```

### 4.7.6. Funktionenzeiger

Variablen und Funktionen sind bei den so genannten VON-NEUMANN-Rechnern im selben Speicher untergebracht und auch als Speicherobjekte verfügbar. Daher ist es in ***C*** auch möglich, Zeiger auf Funktionen zu setzen. Sinn des ganzen ist, dass man auch Funktionen als Argumente übergeben kann, um beispielsweise einen Algorithmus für beliebige Datenobjekte verwenden zu können. Ein Beispiel ist die Sortierfunktion `qsort()`, die neben Zeigern auf die zu sortierenden Elemente auch einen Zeiger auf eine Vergleichsfunktion enthält, die je zwei Elemente vergleicht und für die drei Möglichkeiten kleiner, gleich, größer Werte kleiner, gleich oder größer Null zurückgibt. Die Deklaration der Funktion `qsort()` sieht folgendermaßen aus:

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

Der Name einer Funktion ist immer auch ein Zeiger auf die Funktion; beispielsweise würde `qsort()` die Funktion `strcmp()` als Argument akzeptieren:

```
qsort(charfeld, elementzahl, sizeof(char), strcmp);
```

## 4.8. Überblick über Typ-Deklarationen und -Namen

`int` `int`

`int *` Zeiger auf `int`

`int *[3]` Array von drei Zeigern auf `int`, z.B. `int *iptr[3]`.

`int (*)[3]` Zeiger auf ein Array von drei `int`s, z.B. `int (*i3ptr)[3]`.

`int (*)[*]` ***C***99: Zeiger auf ein Array variabler Länge mit einer unspezifizierten Anzahl von Elementen vom Typ `int`.

`int *()` Funktion, die einen Zeiger auf `int` zurückgibt aber keine Parameterspezifikation hat, z.B. `int *ReturnInt()`, das letztendlich als
`int *ReturnInt(char *blah) {return *char;}` definiert sein könnte.

`int (*)(void)` Zeiger auf eine Funktion mit Rückgabetyp `int` und leerer Parameterliste, z.B. `int (*fptr) (void)`.

`const int *, int const *` Zeiger auf `const`-qualifiziertes `int`, d.h. über diesen Zeiger wird das Objekt, auf das gezeigt wird, garantiert nicht verändert, selbst wenn es veränderbar ist.

`int * const` Konstanter Zeiger auf `int`, d.h. das Objekt, auf das gezeigt wird, ist immer das gleiche und darf über den Zeiger verändert werden.

`int (*const [])(unsigned int, ...)` Feld von unspezifizierter Elementezahl mit Elementtyp konstanter Zeiger auf Funktionen, die `int` zurückgeben und jeweils als ersten Parameter ein Argument vom Typ unsigned int und eine nicht näher spezifierte Anzahl von weiteren Parametern haben.

Funktionen mit variabler Argumentzahl werden erst in Abschnitt 6.6 behandelt; ein Beispiel für den letzten Punkt mit `double` als Rückgabetyp könnte folgendermaßen aussehen:

```
double ArithmetischesMittel (unsigned int n, ...);
double GeometrischesMittel (unsigned int n, ...);
double Median (unsigned int n, ...);
double Maximum (unsigned int n, ...);
double Minimum (unsigned int n, ...);

double (*const StatistikFunktionen[]) (unsigned int, ...)
    = { ArithmetischesMittel, GeometrischesMittel,
        Median, Maximum, Minimum };
```

Sehr hilfreich kann für den Anfang das Programm `cdecl` sein, das Deklarationen von Englisch nach ***C*** und umgekehrt übersetzt.

## 4.9. Strukturen

Will man mehrere (logisch) zusammengehörige Daten von unterschiedlichem oder auch gleichem Typ zusammenfassen, beispielsweise für eine kleine Adressdatenbank einen Adresseintrag, kann man das mit Hilfe des Datentyps `struct` tun:

```
struct AddressEntry { // Namensstil 1
    char surname[NAMELEN];
    char firstnames[NAMELEN];
    short tel[TELLEN];
    char street[NAMELEN];
    short housenumber;
    char add_desc[DESCLEN];
    char city[NAMELEN];
    char postal_code[DESCLEN];
    char country[NAMELEN];
};

struct my_complex { // Namensstil 2
    double re;
    double im;
    int IsReal;
} cplx;

main()
{
    struct AddressEntry addresslist[LISTLEN];
    struct my_complex aplusib, xplusiy;

    aplusib.re = 1.57; aplusib.im = 0.0;
    aplusib.IsReal = 1;

    strcpy(addresslist[0].firstnames,"Heini");
}
```

Der Zugriff auf einzelne Einträge der Struktur erfolgt mittels des Punktes. Liegt ein Zeiger auf die Struktur vor, kann auf die Einträge entweder durch Dereferenzierung des Zeigers oder durch Verwendung der „Abkürzung“ `->` zugegriffen werden:

```
struct AddressEntry *currententry;
// strcpy((*currententry).firstnames,"Heini");
// Kuerzer:
strcpy(currententry->firstnames,"Heini");
```

Auf die Typdefinition kann sofort die Variablendeklaration folgen (dies gilt auch für `union` und Bitfeld, s.u.), im obigen Beispiel ist `cplx` also eine Variable vom Typ `struct my_complex`.

Der Name des Strukturtyps (hier z.B. `my_complex`) wird *Tag* („Etikett“) genannt. Für Tags von Strukturen, Vereinigungen und Aufzählungen (alle zusammen genommen) gilt, dass sie eindeutig sein müssen.

## 4.10. Vereinigungen

Mitunter hat man je nach Kontext unterschiedliche Arten von zusätzlich benötigten Daten. Legt man für jeden Kontext die entsprechenden Daten an, entstehen unter Umständen große Datenstrukturen. Um die Speicherverschwendung zu vermeiden, gibt es die Möglichkeit, die Daten kontextabhängig mit Interpretationen zu versehen, aber sie alle an der gleichen Stelle zu speichern. Dafür gibt es den Datentyp `union`:

```
#define ECHTERBRUCH 1
#define BINAERBRUCH 2

/* Je nachdem, was wir brauchen, halten wir einen
** echten Bruch oder einen Binaerbruch vor
*/

struct rational {
    signed int zaehler;
    signed int nenner;
};

union bruch {
    struct rational echterbruch;
    double binaerbruch;
};

struct mein_bruch {
    char typ; // ECHTERBRUCH, BINAERBRUCH
    union bruch info;
};

int main (void)
{
    struct mein_bruch temp, fraclist[NUMOFOBJ];
    union bruch b1, b2, b3;

    ....

    switch(temp.typ) {
     case ECHTERBRUCH:
        b1.echterbruch = addiere(temp.info.echterbruch,
                                 fraclist[0].info.echterbruch);
```

```
            b2.bineaerbruch = ((double) b1.echterbruch.zaehler)
                              / b1.echterbruch.nenner;
            if (b2.binaerbruch * b1.echterbruch.nenner
                 == (double) b1.echterbruch.zaehler)
            {
                ....
            }
            ....
            break;
         case BINAERBRUCH:
            b1.binaerbruch = temp.info.binaerbruch
                            + fraclist[0].info.binaerbruch;
            b2 = b1;
            ....
            break;
         default:
            printf("%s, %u: Unbekannter Bruchtyp!\n",
                   __func__, (unsigned)__LINE__);
            ....
      }

      ....
   }
```

Je nachdem, um welchen Typ es sich handelt, kann dann entsprechender Code ausgeführt werden.
**Anmerkung:** Die `union` liegt an genau der gleichen Speicheradresse, an der jeder ihrer Einträge startet. Fängt jeder Eintrag der `union` in gleicher Weise an, in obigem Beispiel wären das dann lauter Strukturen, die als erstes Element `char typ;` hätten, so kann der `union` ein generisches Objekt hinzugefuegt werden, das genau diese gemeinsamen Informationen enthält, mittels dessen dann der Typ „von außen“ ausgewertet werden kann.

In diesem Fall ist `bruch` das zur Vereinigung gehörige Tag.

## 4.11. Bitfelder

Für Hardware nahe Programmierung gibt es in ***C*** auch die Möglichkeit, einzelne Bits zusammenzufassen:

```
struct port {
    unsigned int toggle_bit: 1;
    signed   int data_in:    4;
    unsigned int:            1; // unzugaenglich
    unsigned int control:    2;
} myport;

....
```

```
if (myport.control>1) {
    signed info = (signed) myport.data_in;
    if (info<0) {
        ....
    }
    ....
}
```

In Standard-*C* sind als Typen für die Bitfelder nur `unsigned int` und `signed int` zugelassen. Wie dieser Typ im Speicher angeordnet wird, ist Plattform- und Compiler-abhängig, und er sollte deshalb nur dann eingesetzt werden, wenn unbedingt nötig.

## 4.12. (*C* 99-)Initialisierung von Feldern, Strukturen und Vereinigungen

Man kann Feldern, Strukturen und Vereinigungen direkt bei der Deklaration einen Gesamt-„Wert“ zuweisen. Felder können anschließend nie wieder auf einen Rutsch mit einem Wert versehen werden, Strukturen und Vereinigung nur beim Kopieren, d.h. indem man eine Struktur/Vereinigung, die die gewünschten Werte enthält, auf die rechte Seite einer Zuweisung mit = stellt.

Bei der Initial-Definition werden Felder von vorne nach hinten mit den Werten auf der rechten Seite gefüllt, wenn die Werte ausgehen, wird der Rest mit Nullen aufgefüllt (siehe auch Abschnitt 2.3.4), d.h.

```
int feld[7] = { 0, 0, 0, 1};
```

ist gleichwertig zu

```
int feld[] = { 0, 0, 0, 1, 0, 0, 0};
```

Unter *C* 99 sind für diese Art der Initialisierung auch eine variable, erst zur Laufzeit bekannte, Feldgröße und nichtkonstante Einträge möglich. Daneben gibt es aber auch noch die Möglichkeit der designierten Initialisierung, mit der sich gezielt einzelne Feldelemente bei der Initial-Definition ansprechen lassen, z.B. im obigen Fall

```
int feld[7] = { [3] = 1};
```

Alle nicht angesprochenen Feldelemente werden auf Null gesetzt.

In ähnlicher Weise gilt das auch für Strukturen: Unter *C* 89 muss eine Initial-Definition zur Struktur passen, etwa im Bruch-Beispiel

```
struct rational r1    = {1, 2};
struct mein_bruch mb1 = { ECHTERBRUCH, {2, 3} };
```

Bei einer Vereinigung kann nur das erste Element initialisiert werden, d.h.

```
union bruch b1        = {5, 7};
```

ist möglich, aber

```
union bruch b2        = {7.5};
struct mein_bruch mb2 = { BINAERBRUCH, {.38} };
```

sind beide unter ***C*** 89 falsch.

Mit der designierten Initialisierung ist es möglich, gezielt einzelne Elemente einer Struktur bzw. ein bestimmtes Element einer Vereinigung zu initialisieren:

```
struct rational r1    = {.zaehler = 1, .nenner = 2};
struct rational r2    = {.nenner = 1, .zaehler = 2};
union bruch b2        = {.binaerbruch = 7.5};
struct mein_bruch mb3 = {.info.echterbruch.nenner = 1} };
```

## 4.13. Aufzählungen

Mitunter ist es ganz praktisch, wenn man nicht nur Zahlen in einer Abfrage hat, sondern weiß, welche Interpretation sich hinter ihnen „versteckt“. Eine Möglichkeit ist die Verwendung von symbolischen Konstanten; dann muss man sozusagen beim Hochzählen aufpassen, dass nichts doppelt verwendet wird bzw. beim Einfügen alles weiterschieben. In ***C*** können ähnlich wie Strukturen und Vereinigungen auch Aufzählungen definiert werden, die dann bestimmte Werte und ihre Wertinterpretation haben.

***C***-Aufzählungen mit `enum` beginnen bei 0 und steigen dann mit jedem weiteren Wert um 1. Man kann aber auch Werte auslassen, indem man direkt Werte zuweist. Die in einer Aufzählung enthaltenen Werte heißen Aufzählungskonstanten. Vorsicht ist geboten, dass man die Wertinterpretation nicht mit der Bezeichnung in der Aufzählung durcheinanderwirft. Hierfür sollte man gleich von vornherein eine Funktion vorsehen, die einem Wert die entsprechende Interpretation zuweist.

```
enum Woche { Montag, Dienstag, Mittwoch, Donnerstag,
             //   0,        1,        2,          3,
             Freitag, Samstag, Sonntag };
             //    4,       5,       6

enum Geldwerte { nix, eincent, zweicent, fuenfcent=5,
                 zehncent=10, zwanzigcent=20,
                 fuenfzigcent=50, eineuro=100 };

int Name (enum Woche wochentag, char *name)
{
    switch (wochentag) {
     case Montag:
        strcpy(name,"Montag");
        break;
     case Dienstag:
        ....
```

```
        default:
           name = NULL;
           return(__LINE__);
       }
       return(0);
   }

   int main ()
   {
       enum Woche week;
       char name[10];

       ....
       switch (week) {
        case Montag:
        case Dienstag:
        case Mittwoch:
        case Donnerstag:
        case Freitag:
           if( Name(week, name) )
               printf("Fehler in Name(), Zeile: %d",
                      Name(week, name));
           ....
       }
       ....
   }
```

Ein Beispiel für die Verwendung designierter Initialisierung in Verbindung mit einer Aufzählung für bessere Wartbarkeit findet sich im Beispielprogramm für Sortieralgorithmen, siehe Seite 173.

**Anmerkung:** Für Variablen eines Typs `enum` *tag* gilt *nicht*, dass sie nur auf die Aufzählungskonstanten-Werte beschränkt sind, sondern nur, dass sie „groß genug“ sind, um alle möglichen Aufzählungskonstanten darzustellen!

## 4.14. Konstanten

Den einzelnen Datentypen werden oft am Anfang konstante Werte zugewiesen. Konstanten können aber auch direkt bei Funktionsaufrufen übergeben werden, dann ist es wichtig zu wissen, als was sie interpretiert werden. Die meisten Konstanten sind bereits aufgetaucht, sie sollen aber hier noch einmal erwähnt werden:

- `char`: Zeichen in Hochkommata, z.B. `'a'`, `'0'`, `'\n'`.
- `wchar_t`/Multibyte-Zeichen: Hier wird ein `L` wie Long vorangestellt, um das Zeichen von einem normalen `char` abzugrenzen, z.B. `L'a'`, `L'0'`, `L'\n'`.

- Integer-Datentypen: Zahlen werden meistens als `int`-Konstanten des kleinstmöglichen Datentyps interpretiert. Mit Hilfe eines nachgestellten `L` bzw. `U` werden `long`- und `unsigned`-Konstanten gekennzeichnet, z.B. `int`: `10`, `-123`; `unsigned`: `30456U`; `long`: `1L`, `-2111333890L`; `unsigned long`: `5UL`; `long long`: `37LL`...

- Hexadezimale/oktale Konstanten: Hier handelt es sich ebenfalls um ganzzahlige Datentypen, die aber in einem anderen Zahlensystem angegeben werden. Meistens enthält heute ein Byte acht Bit, dezimal 0 bis 255. Durch die Darstellung im hexadezimalen (Sechzehner-)System können Bytes dann durch zweistellige Zahlen dargestellt werden, deren Ziffern jeweils vier Bit (ein *Nibble*) umfassen. Die Zahlen 10 bis 15 werden durch die Ziffern `A` bis `F` dargestellt. Hexadezimale Konstanten werden durch vorangestelltes `0x` gekennzeichnet. Aus ähnlichen Gründen gibt es auch Konstanten im oktalen (Achter-)system; sie werden durch eine vorangestellte `0` gekennzeichnet. Zeichen können ebenfalls hexadezimal bzw. oktal angegeben werden durch `'\x`*N*`'` bzw. `'\`*N*`'`, wobei *N* die hexadezimale bzw. oktale Zahlendarstellung ist. Der Stringterminator `'\0'` ist also ebenfalls eine oktale Konstante. Die dezimale Variante wäre `(char) 0`. Beispiele: `077` ($= 7 \cdot 8 + 7 \cdot 1 = 63$), `0x8` ($= 8 \cdot 1 = 8$), `0xC3` ($= 12 \cdot 16 + 3 = 195$)
  Hexadezimale und oktale Konstanten sind für bitweises Arbeiten praktisch, aber auch eben für angepasste Interpretationen: Gibt man Farben im RGB-System ein (z.B. bei der Erstellung von Webseiten), so wird für Rot, Grün und Blau jeweils ein gleich großer Werte-Raum verwendet, i.d.R. vier, sechs oder acht Bit (ergibt 4096, 262144 oder 16.8 Mio Farben). Mit der Angabe für achtbittige RGB-Werte „Farbe 8735366“ lässt sich wenig anfangen, aus „Farbe 0x854A86“ lässt sich dagegen ablesen, dass es sich um eine Farbe handelt, deren Rot- und Blau-Anteile etwa doppelt so hoch sind wie der Grün-Anteil und die insgesamt eher dunkel ist (0x000000: Schwarz; 0xFFFFFF: Weiß).

- Fließkommazahlen: Wann immer ein Punkt oder ein Exponent in einer Zahl auftaucht, wird sie als Fließkommazahl interpretiert. Hier ist der Standardtyp nicht der kleinste, `float`, sondern `double`. `float`-Konstanten werden durch nachgestelltes `F` gekennzeichnet, `long double`-Konstanten durch nachgestelltes `L`. Exponenten werden durch `e` bzw. `E` gekennzeichnet und eine Zahl *A*e*B* wird als $A \cdot 10^B$ interpretiert. Beispiele: `0.37F`, `3e12F`, `4.15`, `2.18e-35`, `1.0L`.

- Strings: Konstante Strings werden durch obere Anführungszeichen gekennzeichnet, z.B. `"Das ist ein toller String"`, und sind auch unter dem Namen *Stringliteral* bekannt; eine Besonderheit ist, dass ein Stringliteral vom Typ **`char *`** ist, aber trotzdem nicht modifizierbar – was man sonst eher von **`const char *`** (siehe Abschnitt 4.15.2) erwarten würde; ein häufiger Fehler in dieser Hinsicht ist, bei der Deklaration nicht aufzupassen:

```
/* Initialisiere string1 durch "Mein schoener String" */
char string1[] = "Mein schoener String";
/* string2 zeigt auf das Stringliteral */
char *string2  = "Mein schoener String";

/* Wandle Kleinbuchstaben in string1 in Grossbuchstaben um */
AllesGross(string1);
/* Fehler! */
AllesGross(string2);
    /* Stringliterale duerfen nicht modifiziert werden */
```

```
/* Jetzt zeigt string2 woanders hin. */
string2 = "Hallo, Welt\n";
/* Fehler! */
string1 = "Hallo, Welt\n";
    /* Korrekt: strcpy(string1, "Hallo, Welt\n"); */
```

## 4.15. Weitere Typ-Attribute

Neben dem eigentlichen Datentyp gibt es noch weitere Attribute, die das Speicher- und Variablenverhalten angeben: Speicherqualifizierende und typqualifizierende Schlüsselworte. Bei der Deklaration gilt die Aufrufreihenfolge

*Speicherqualifizierer Typqualifizierer Datentyp Name (Definition);*

### 4.15.1. Speicherklasse

**Automatische Variablen** Ohne weitere Angaben sind alle Variablen automatische (`auto`-) Variablen, d.h. sie werden bei jedem Eintreten in ihren Gültigkeitsbereich – Global, Datei, Funktion oder Block – angelegt und nach Verlassen wieder gelöscht. Das Schlüsselwort `auto` braucht nie angegeben zu werden.

**Statische Variablen** `static`-Variablen werden beim ersten Eintreten in ihren Gültigkeitsbereich angelegt und gegebenenfalls mit einem Wert initialisiert, beim Verlassen aber nicht zerstört. Wann immer man in den Gültigkeitsbereich wieder eintritt, haben sie den gleichen Wert, den sie beim letzten Verlassen hatten. `static`-Variablen existieren nur in dem Gültigkeitsbereich, in dem sie deklariert sind. Dies gilt insbesondere für Variablen (und damit auch Funktionen), die dateiweit gültig sind.

**Externe Variablen** Es besteht keine Möglichkeit, `static`-Variablen global gültig zu machen, bei auf Datei-Ebene definierten `auto`-Variablen dagegen schon: Will man in einer anderen Datei auf sie zugreifen, deklariert man in dieser Datei die Variable (oder Funktion) als `extern` und kann fortan auf sie zugreifen. Das `extern`-Schlüsselwort besagt, dass die Variablen an anderer Stelle deklariert sind.

**Register-Variablen** Es gibt für viele Algorithmen Variablen, die kritisch sind, weil oft auf sie zugegriffen und mit ihnen gearbeitet wird. Am schnellsten funktioniert der Zugriff auf eine Variable dann, wenn sie direkt im Prozessor gespeichert ist. Daher wurde, ursprünglich nur für Ganzzahltypen und `char`-Variablen sowie Zeigervariablen die Speicherklasse `register` eingeführt, die dem Compiler sagte, dass bei der Erzeugung des Assembler-Codes diese Variable stets in einem Prozessor-Register gehalten werden sollte, wenn möglich. Mittlerweile wurde diese Definition etwas „aufgebohrt“: Mit `register` teilt der Programmierer mit, dass er die entsprechende Variable (von beliebigem Datentyp) für kritisch hält und sie bevorzugt behandelt werden soll. `register` kann auch mit `static` und `extern` kombiniert werden.

```
/* Datei 1 */

int globalvar = 0; // Deklaration und Initialisierung

static int func1()
{
    static int counter=1;

    if(counter>10)
        globalvar = globalvar*(counter-10);

    return(++counter);
}


/* Datei 2 */

extern int globalvar; // Darf nur deklariert werden


extern int func1(); // FEHLER: Nur in Datei 1 bekannt!

void func2 (int parm)
{
    register int pivot;
    extern char spaeterdef;

    switch(globalvar/parm) {
        ....
    }
    ....
}

char spaeterdef = 'A';
```

### 4.15.2. Typklasse

**Konstante Variablen** Mit dem Schlüsselwort `const` werden Variablen für den entsprechenden Gültigkeitsbereich als konstant, d.h. unveränderlich, gekennzeichnet. Das hat den Sinn, dass man sich selbst vor einer versehentlichen Änderung der Variable schützen kann, z.B. wenn man einen Zeiger auf die Variable übergibt. Beispielsweise ist der zu kopierende String in der Deklaration von `strcpy()` vom Typ `const char *`, d.h. der Zeiger zeigt auf ein Objekt vom Typ `const char`.

Beim Arbeiten mit Stringliteralen ist es sinnvoll, diese entweder zur Initialisierung von Feldern zu verwenden oder mit entsprechenden Zeigern zu arbeiten, damit man gewarnt wird, wenn man

versucht, sie zu überschreiben. Im Beispiel von Seite 77 wäre das:

```
const char *string2  = "Mein schoener String";

AllesGross(string2);
    /* Compiler maekelt, weil wir an eine String-
    ** modifizierende Funktion einen Zeiger auf etwas
    ** nicht modifizierbares übergeben. */
```

Ein konstanter Zeiger auf ein Objekt vom Typ `char` wird als `char *const` deklariert, auf ein konstantes `char`-Objekt dementsprechend als `const char *const`.

**Flüchtige Variablen** In der Regel kann der Compiler davon ausgehen, dass nur das von ihm erzeugte Programm Zugriff auf Variablen hat und dementsprechend für die Optimierung beliebigen semantisch richtigen Code erzeugen. Kann sich eine Variable aber auf nicht definierte Weise ändern, darf der Compiler nicht auf sie verzichten bzw. sie ignorieren, weil er meint, ihr Wert ändere sich nicht von einem Schritt des Programmes zum nächsten. Solche Variablen können etwa Werte an festen Adressen sein, auf die bestimmte Hardware- oder Software-Komponenten genauso zugreifen dürfen wie das eigene Programm, z.B. würde man einen Read-Only-Port mit der Adresse `0x0000A040` folgendermaßen deklarieren:

```
const volatile unsigned *roport = \
    (const volatile unsigned *) 0xA040;
```

Die Typ-Umwandlung (→ Abschnitt 4.16) ist notwendig, damit die Adresse nicht als Zahl wahrgenommen wird.

**Restringierte Zeiger** Je mehr Informationen ein Compiler über die verwalteten Objekte hat, desto besser kann er den Code optimieren. Bestimmte Algorithmen für Zeigeroperationen lassen sich nur dann verwenden, wenn die Zeiger sich gegenseitig nicht ins Gehege kommen, also nie auf die gleiche Adresse zeigen können. Mit Hilfe des in ***C***99 hinzugekommenen Schlüsselwortes `restrict` ist es möglich, diese Information zu geben. Ein durch `restrict` qualifizierter Zeiger ist der erste Zeiger, der auf ein Objekt zeigt, alle anderen Zeiger basieren auf ihm. Das ist z.B. im Fall von per `malloc()` allokiertem Speicher möglich. Hat man zwei verschiedene `restrict`-qualifizierte Zeiger, dann können sich die Objekte, auf die sie zeigen, nicht überlappen.

### 4.15.3. Charakteristiken von Variablen und Funktionen

Jeder *Identifier* (Variable, Funktion, Sprungmarke, Tag, Struktur-, Vereinigungs- oder Aufzählungselement) in einem ***C***-Programm hat vier primäre Attribute/Charakteristiken:

1. *Gültigkeitsbereich* (bereits in Abschnitt 1 besprochen).
2. *Bindung*
3. *Speicherdauer*

Das Schlüsselwort `static` hat sowohl im Bezug auf Bindung als auch im Bezug auf Speicherdauer eine Bedeutung[9].

Die drei Typen von Bindung sind *extern*, *intern* und *keine*. Objekte, die innerhalb irgend eines Blocks definiert sind, haben keine Bindung. Variablen und Funktionen, die auf Dateiebene definiert sind, haben von Haus aus externe Bindung, d.h. man kann auf sie durch ihren Namen aus anderen Übersetzungseinheiten zugreifen (indem man sie per `extern` bekannt macht).
Wird eine Variable oder Funktion, die auf Dateiebene definiert ist, mit dem Schlüsselwort `static` qualifiziert, dann ändert sich die Bindung von extern auf intern: Ihr Name ist nach „außen“ nicht sichtbar, auf sie kann nicht aus anderen Übersetzungseinheiten durch ihren Namen zugegriffen werden (sehr wohl aber über ihre Adresse).

**Beispiel:** Definiert man auf Dateiebene

```
int x;

int funktion (int param)
{
    return x + param;
}
```

dann haben `x` und `funktion` externe Bindung. (Funktionsparameter leben nur innerhalb der Funktionsdefinition, d.h. `param` hat keine Bindung.) Will eine andere Datei bzw. Übersetzungseinheit auf `x` zugreifen, so braucht sie eine Deklaration der Form

```
extern int x;
```

Für Funktionenprototypen kann man das Schlüsselwort `extern` zwar auch benutzen, es ist aber nicht notwendig, weil Prototypen genau wie Funktionsdefinitionen standardmäßig externe Bindung haben

Definiert man dagegen

```
static int x;

static int funktion (int param)
{
    return x + param;
}
```

auf Dateiebene, dann haben `x` und `funktion` interne Bindung.

Die dritte Attributskategorie ist die Speicherdauer; es gibt drei Typen: *statisch*, *automatisch* und *alloziert*. Alle Variablen, die auf Dateiebene definiert sind, haben statische Speicherdauer, d.h. sie existieren und haben einen Startwert, bevor `main` beginnt, und existieren, bis das Programm endet. Alle Variablen, die innerhalb eines Blocks leben und nicht mit einer Speicherklasse versehen

[9] Unter ***C***99 hat `static` eine weitere Bedeutung in Verbindung mit Feldern als Funktionsparametern erhalten, die aber nichts mit der Qualifizierung von Variablen zu tun hat, siehe Abschnitt 6.7.

sind (bzw. mit einem der Schlüsselworte `auto` oder `register` versehen sind), haben automatische Speicherdauer; sie beginnen ihre Existenz, wenn die Programmausführung in ihren Block eintritt, und beenden sie, wenn der Block wieder verlassen wird. Sie haben anfangs keinen bestimmten Wert, wenn sie nicht bei der Deklaration gleich initialisiert werden. Werden Variablen, die innerhalb eines Blocks leben, mit `static` qualifiziert, dann haben sie statische Speicherdauer: Genau wie Variablen auf Dateiebene existieren sie während der gesamten Programmlaufzeit und haben einen Anfangswert, bevor `main` ausgeführt wird. Sie behalten ihren aktuellen Wert, wenn ihr Block verlassen wird. Die dritte Art von Speicherdauer, alloziert, gehört zu Speicher, der aus einem Aufruf von `malloc()`, `calloc()` oder `realloc()` stammt. Dieser Speicher besteht so lange, bis er durch einen Aufruf von `free()` bzw. indirekt durch einen Aufruf von `realloc()` freigegeben wird.

Ein name space (wörtlich Namensraum) bezeichnet die Bereiche, in denen ein Name etwas eindeutig identifiziert:

- Sprungmarken (siehe Abschnitt 9.3.4)
- Tags von Strukturen, Vereinigungen und Aufzählungen (alle zusammen, d.h. eine Struktur und eine Vereinigung dürfen nicht den gleichen Tag haben).
- Elemente von Strukturen und Vereinigungen: Es ist möglich, dass zwei verschiedene Strukturen (Vereinigungen) unter ihren Elementen welche gleichen Namens haben, da jedes Element durch die übergeordnete Struktur (Vereinigung) eindeutig gemacht wird; eine Struktur (Vereinigung) darf dagegen keine zwei Elemente gleichen Namens besitzen.
- Alle anderen Identifier, auch gewöhnliche Identifier genannt (also Funktions- und Variablennamen, sowie Namen von Aufzählungskonstanten).

**Beispiel:** Der Identifier `x` wird hier verschiedentlich mehrfach verwendet, mal legal, mal nicht:

```
struct x {
    int x; // kein Widerspruch (struct x).x
} x; // kein Widerspruch

struct X {
    int x; // kein Widerspruch (struct X).x
    float x; // geht nicht (struct X).x bereits vorhanden
} x; // geht nicht, Variablenname bereits vorhanden

union x { // geht nicht, Tag bereits vorhanden
    ....
};

union y {
    int x; // kein Widerspruch (union y).x
} x; // geht nicht, Variablenname bereits vorhanden

int x; // geht nicht, Variablenname bereits vorhanden
void x () // geht nicht, bereits als Variable vorhanden
```

```
{
    ....
}

void Y ()
{
    int x; // moeglich, da anderer Gueltigkeitsbereich,
           // shadowing von struct x x;
    ....
x: // kein Widerspruch, Sprungmarke x:
    ....
x: // geht nicht, Sprungmarke x: bereits vorhanden
    ....
}
```

## 4.16. Typumwandlung

Wenn man zwei Ganzzahl-Variablen dividiert, kommt wieder eine Ganzzahl-Variable heraus. Will man das wirkliche Ergebnis dieser Division (in Maschinengenauigkeit) wissen, muss man das mittels einer Typ-Umwandlung (*typecasting*) bekanntgeben. Dies geschieht mittels *(Zieltyp) Ausgangsobjekt*, in unserem Beispiel sieht das dann folgendermaßen aus:

```
int z=1, n=2;
int i_erg;
double erg;

i_erg = z/n; // 0
erg = (double) i_erg; // 0.0
erg = (double) (z/n); // 0.0
erg = ((double) z)/((double) n); // 0.5
erg = ((double) z)/n; // 0.5
```

Die letzte Regel ist nicht völlig klar: Hier handelt es sich um ein Beispiel von Typ-Promotion, d.h. um die Rechnung durchführen zu können, verwendet der Rechner den „größeren“ von beiden Datentypen für beide Variablen, es wird also intern ein Typecast durchgeführt. `double` ist in dem Sinne größer als `int`, dass `double` in der Regel einen größeren Zahlenbereich umfasst und `int`s (mit oder ohne Rundungsverluste(n)) darstellen kann. Wegen der Operatorpriorität kann man die Zeile auch so

```
erg = (double) z/n; // 0.5
```

schreiben.

Typ-Umwandlung spielt bei der Arbeit mit Zeigern eine Rolle: Viele Funktionen erwarten lediglich irgendeinen Zeiger als Argument bzw. geben nur irgendeinen Zeiger zurück. „Irgendein“ wird durch den Zeigertyp `void *` abgedeckt. Ein Beispiel für eine solche Funktion ist `malloc()`, das einen `void *` als Anfang des allokierten Speichers zurückliefert, der dann auf den gewünsch-

ten Zeigertyp gecastet wird. Das erfolgt automatisch und sollte in ***C*** (ganz im Gegensatz zu ***C++***) nicht explizit erfolgen.

Explizite Typecasts sollte man in ***C*** nur dann einsetzen, wenn es nötig ist. In seltenen Fällen kann man damit aber auch für sich selbst markieren, dass man weiß, um welchen Typ es sich handelt, und dieser Typ auch gewollt ist, bzw. dass man etwas bewusst nicht macht. Ein Beispiel für letzteres ist der Rückgabewert von `scanf`– eigentlich sollte man immer überprüfen, ob tatsächlich ein entsprechendes Objekt eingelesen werden konnte; ist man sich aber ganz sicher, dass das nicht notwendig ist, könnte man

```
(void) scanf(....);
```

schreiben.

Umgekehrt kann es sein, dass eine Funktion gar nicht wissen muss, was für ein Objekt sie bearbeitet, sondern nur einen Zeiger braucht und die Größe des Objektes, auf das dieser Zeiger zeigt. Ein Beispiel dafür ist die Funktion `qsort()`, die eine Menge von Objekten sortiert und dazu als Parameter ein Feld dieser Objekte dessen Größe, die Objektgröße sowie eine Funktion, die zwei Objekte vergleicht, erwartet:

```
#include <stdlib.h>

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

Manchmal gibt es auch Fehlermeldungen über ein nicht passendes Argument bei einem Funktionsaufruf, weil z.B. das übergebene Objekt nicht auf `const` *typ* gecastet wurde.

Zurück zur Promotion von Ganzzahlen und Fließkommazahlen: Ganzzahl-Rechnungen mit `char`, `short` und `int` werden in Standard-***C*** mit `int`s durchgeführt und dann auf den Zieldatentyp gecastet; wenn der Zahlenbereich nicht ausreicht, liefert die Operation ein falsches Ergebnis. Hierbei ist die Umwandlung nicht vorzeichenerhaltend, sondern rein werterhaltend, d.h. ein **`unsigned char`**-Wert, der durch `int`, darstellbar ist, wird auch in `int` umgewandelt und nicht in **`unsigned int`**. Wenn `long` sich von `int` unterscheidet, kann durch einen typecast auf `long` ausgewichen werden. Genauso werden `float`- und `double`-Rechnungen in doppelter Genauigkeit (also mit `double`s) ausgeführt und dann gecastet bzw. können nötigenfalls in `long double` ausgeführt werden. Je nach Compiler bzw. je nach `FLT_EVAL_METHOD` in `<float.h>` kann es auch sein, dass alle Operationen in `long double` ausgeführt werden[10]. Übergibt man an eine Funktion, die einen größeren Ganzzahl-Datentyp erwartet, einen kleineren Ganzzahl-Datentyp, so wird die Erweiterung ohne Nachfrage durchgeführt. Hat man zwei Operanden unterschiedlichen Typs, so wird der kleinere bzw. Ganzzahltyp entsprechend umgewandelt.

Für Funktionen, deren Argumente nicht a priori bekannt sind (weil der Prototyp, der sie bekannt macht, ohne Argumente angegeben wurde, oder für Funktionen mit variablen Argumenten wie `printf` und `scanf`), wird automatisch jeder kleinere Ganzzahl-Typ auf `int` und `float` auf

[10] *gcc* weist hier leider fehlerhaftes Verhalten auf: Selbst ein expliziter Cast auf `double` hilft unter Umständen nicht, dass ein `long double`-Wert zu einem `double`-Wert wird. Benötigt man das unbedingt, so sollte man den Code mit den Optionen `-ffloat-store`, `-mfpmath=sse`, `-msse2` übersetzen, die jedes Fließkomma-Zwischenergebnis in der entsprechenden Genauigkeit abspeichert, was aber den Code sehr langsam macht.

`double` erweitert. Ist man sich nicht ganz sicher, dann ist es sicherer, die Argumente solcher Funktionen zu casten.

## 4.17. Eigene Typen

Hat man einen komplexen Datentyp erstellt, möchte man unter Umständen nicht jedes Mal für Variablen des entsprechenden Typs den ganzen Typnamen schreiben. Die naheliegendste Möglichkeit ist wiederum die Abkürzung per symbolischer Konstante:

```
#define MYSTRUCT    struct mystruct
#define MYSTRUCTPTR struct mystruct *

struct mystruct {
    ....
};

int main (void)
{
    int i, j, k;
    MYSTRUCT s1, s2;
    MYSTRUCTPTR sptr1, sptr2;
}
```

Dummerweise ist `sptr2` jetzt kein Zeiger, weil durch die Textersetzung nur vor `sptr1` ein `*` steht. Es gibt aber die Möglichkeit, eigene Datentypen zu definieren mittels *typedef Typ Neuername;*:

```
struct mystruct {
    ....
};

typedef struct mystruct   MYSTRUCT;
typedef struct mystruct * MYSTRUCTPTR;

int main (void)
{
    int i, j, k;
    MYSTRUCT s1, s2;
    MYSTRUCTPTR sptr1, sptr2;
}
```

Die explizite Benennung der Struktur ist nicht unbedingt nötig. Nachdem Tags aus einem anderen namespace stammen, ist es auch möglich, dem neuen Typ den gleichen Namen zu geben wie dem Tag.

```
typedef struct mystruct mystruct;
```

Nach der `typedef`-Anweisung ist der Typ bekannt und kann seinerseits auch wieder in einer `typedef`-Anweisung verwendet werden.

```
typedef struct {
    ....
} MYSTRUCT;

typedef MYSTRUCT * MYSTRUCTPTR;
```

Da Strukturen formal verwendet werden können, bevor sie definiert worden sind, kann man die Typdefinition vor die Strukturdefinition stellen. Es ist auch möglich, Funktionenzeiger-Typen zu definieren:

```
typedef double (*LMShapeFunctionProcPtr) \
                            (int, int, double *, int);
```

Nützlich ist das vor allem, für komplexe Typen, wie im Beispiel von Seite 70:

```
double (*const StatistikFunktionen[]) (unsigned int, ...)
    = { ArithmetischesMittel, GeometrischesMittel,
        Median, Maximum, Minimum };
```

ist schwerer zu verdauen als

```
typedef double (* StatFunPtr) (unsigned int, ...);

const StatFunPtr[] = { ArithmetischesMittel,
    GeometrischesMittel, Median, Maximum, Minimum };
```

Man kann sich die Syntax von `typedef` relativ einfach merken, da sie die gleiche ist wie die von Speicherklassen bei Variablen-Deklarationen, d.h. man ersetze in Gedanken `typedef` durch z.B. `static`.

Von der Standard-Bibliothek definierte Typen werden mit einem angehängten `_t` kenntlich gemacht, z.B. `wchar_t`, `size_t`, `int32_t`. Einzige Ausnahme sind hier die typedefs `bool`, `complex` und `imaginary`.

Das einführende Beispiel ist mit Vorsicht zu genießen: In der Regel ist es besser, Zeiger nicht zu verstecken, da sonst eher einmal Flüchtigkeitsfehler auftreten. In ***C***89-Code findet man häufig Typen, die zu den Ganzzahldatentypen mit festgelegter (Mindest-)Bitzahl in ***C***99 geführt haben (z.B. `u8`, `int16`, `byte`, `word`).

Genau betrachtet kann man nur mit Hilfe von `struct` neue Typen erzeugen – `typedef` sorgt nur dafür, dass sie über einen bestimmten Namen ansprechbar sind. Für komplexe Typdefinitionen macht `typedef` diese oft erst auch lesbar.

## 4.18. Überblick: Datentypen und Speicher

Jede Speicherzelle des Rechners umfasst ein Byte. Der Prozessor kann intern Ganzzahl-Variablen einer bestimmten Größe auf einmal speichern. Diese Größe ist ein Vielfaches der Byte-Größe.

Ursprünglich ist vorgesehen, dass `int` genau diese Größe hat. Für gewöhnlich sind Adressen ebenfalls von dieser Größe, so dass der Rechner maximal so viele Bytes adressieren kann, wie im `int`-Datentyp dargestellt werden können. In der Regel wird die Größe des adressierbaren Speichers durch andere Hardware-Komponenten beschränkt.

Beispiel: Für eine 32-Bit-Architektur mit 8-Bit-Bytes ergibt sich eine Integer-Größe von 4 Bytes und eine maximale Speichergröße von $2^{32}$ Bytes, also 4 Gigabyte. Ist der Speicherbus auf z.B. 24 Bit beschränkt, können nur $2^{24}$ Byte, also 16 Megabyte adressiert werden. Für 64-Bit-Architekturen ergibt sich das Problem, dass viele Programmierer sich sehr stark an die Zuordnung „ein `int` ist 32 Bit“ gewöhnt haben und deshalb viele Programme nicht mehr in der gewohnten Weise laufen würden. Daher ist die `int`-Größe stark compilerabhängig. Aber auch sonst ist es sinnvoll für größere Programme, erst einmal die Möglichkeiten und Grenzen zu überprüfen. Dies geschieht, indem man die in `<limits.h>` und`<float.h>` definierten Werte wie z.B. `INT_MAX` bzw. `DBL_MIN` ausliest und gegen die Annahmen im eigenen Programm überprüft.

Es gibt auch Systeme, auf denen die Größe von Zeigern für verschiedene Objekte verschieden ist.

Der am Anfang dieses Kapitels eingeführte `sizeof`-Operator liefert die Größe eines Objektes in Byte. Die Rückgabe-Größe ist vom Typ `size_t`, der per `typedef` in `<stdlib.h>` definiert wird. Das hat den Vorteil, dass, wenn sich von Rechner zu Rechner die Größe des adressierbaren Speichers ändert, man nicht alle Funktionen umwerfen muss. In der Regel ist `size_t` etwas in der Art von `unsigned`; das spielt dann eine Rolle, wenn tatsächlich der komplette theoretisch adressierbare Speicher auch im Rechner vorhanden ist - dann verliert man, falls man statt `size_t` einfach `int` verwendet, ein Bit bei den Adressen und damit die Hälfte der Adressen...

Oft gibt es eine zusätzliche Beschränkung für Datentypen, die größer als ein Byte sind: Um ein schnelles Laden vom Speicher in den Prozessor zu ermöglichen, müssen die Variablen unter Umständen auf den Wort- oder Integer-Grenzen anfangen, d.h. auf dem erwarteten Standard-System (8-Bit-Bytes, 4-Byte-Integers) müssten Zwei-Byte-Datentypen (z.B. `short`) dann auf geraden und Vier-oder-mehr-Byte-Datentypen (`int`, `double`) auf durch vier (`sizeof(int)`) teilbaren Adressen starten. Das gilt auch für und innerhalb von Strukturen; eine Anordnung der Art `char-int-char-double-char` füllt (auf das nächste Vielfache von vier aufgerundet) dann 24 Byte, während die Umordnung `double-int-char-char-char` nur 16 Byte belegt. Eine einfache Regel für effiziente Speicherausnutzung von Strukturen ist daher „ordne die verwendeten Datentypen von groß nach klein“, sofern das möglich ist (in Verbindung mit Vereinigungen könnte ein bestimmter Teil vorgeschrieben sein).

Abschließend sind in Tabelle 4.1 noch einmal alle Datentypen in Verbindung mit ihrem Speicheraufwand dargestellt.

## 4.19. Verkettete Listen, Bäume, Stapel, Schlangen

Neben den elementaren ***C***-Datentypen gibt es noch einfach zu realisierende andere (nicht im Sprachumfang von ***C*** enthaltene) Datentypen für die sinnvolle Verarbeitung von Daten, insbesondere bei dynamischer Datenanzahl. Eine gute Einführung hierzu findet sich in [Sed92], die neueren englischen Auflagen ([Sed97, Sed01]) sind unter Umständen ausführlicher.

Ein Element einer *verketteten Liste* enthält neben den eigentlichen Daten noch einen (einfach verkettet, siehe Abbildung 4.2) oder zwei (zweifach verkettet, siehe Abbildungen 4.3 und 4.4)

| **Typ** | **Größe** | **typische Werte /Byte** | **Minimal-werte/Bit** |
|---|---|---|---|
| `char` | `sizeof(char)` | 1 | 8 |
| `unsigned char` | `sizeof(unsigned char)` | 1 | 8 |
| `short` | `sizeof(short int)` | 2 | 16 |
| `int` | `sizeof(int)` | 4 | 16 |
| `long` | `sizeof(long int)` | 4 | 32 |
| `long long` | `sizeof(long long int)` | 8 | 64 |
| `float` | `sizeof(float)` | 4 | – |
| `double` | `sizeof(double)` | 8 | – |
| `long double` | `sizeof(long double)` | $10, 12, 16$ | – |
| Zeiger auf *Typ* | `sizeof(`*Typ* `*)` | 4 | – |
| `_Bool` | `sizeof(_Bool)` | 1/4 | 8 |
| `_Complex` | `sizeof(_Complex)` | 16 | – |
| `int Array[N]` | $N*$`sizeof(int)` | $N*4$ | – |
| `float Array[N][M]` | $N*M*$`sizeof(float)` | $N*M*4$ | – |
| `struct { int index; char typ; double x[3]; double xr[2]; } beispiel` | `sizeof(beispiel)` | $4+4+24+16$ | – |

**Tabelle 4.1.:** Datentypen und Speichergrößen; für Fließkommazahlen gibt es keine offensichtlichen Mindest-Bitzahlen sondern nur Genauigkeitsvorgaben in Dezimalstellen und Exponentenbereiche, siehe auch Abschnitt 4.5.

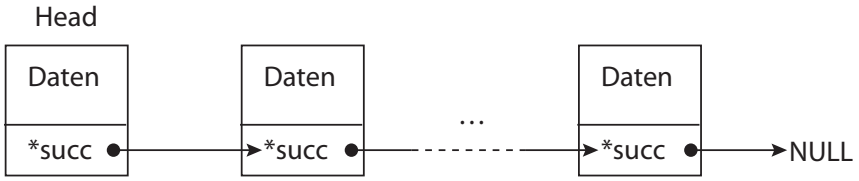



**Abbildung 4.2.:** Einfach verkettete Liste (Variante A)

Zeiger auf das Nachfolgerelement bzw. auf Nachfolger und Vorgänger. Gibt es keinen Nachfolger (Vorgänger), ist der Zeigerwert `NULL`. Das einzige, was man braucht, ist ein Zeiger auf das erste Element bzw. auf ein Kopf-Element, das auf das erste Element zeigt. Letzteres ist praktischer beim Einfügen und Löschen von Elementen; in ähnlicher Weise kann ein Ringschluss auf das Kopf-Element oder das Einführen eines Abschluss-Elementes sinnvoll sein:

```
struct linkedlist {
    struct linkedlist *succ;
    // Auskommentieren fuer einfach verkettete Liste:
    struct linkedlist *pred;

    // Hier beginnen die eigentlichen Daten
    ....
};

typedef struct linkedlist List;

/* Listenkopf/abschluss
**
** Variante A: Nachfolger/Vorgaenger ist immer NULL am Anfang,
**             das letzte Element hat keinen Nachfolger.
**
** Variante B: Head ist sein eigener Nachfolger (/Vorgaenger),
**             es gibt eine Art Ringschluss. Das letzte Element
**             hat Head als Nachfolger.
**
** Variante C: Head hat Tail als Nachfolger (Tail hat Head als
**             Vorgaenger), die anderen Zeiger sind NULL. Das
**             letzte Element hat Tail als Nachfolger.
**
*/

List Head = { // Variante A, B, C
    NULL; // Initialiserung-> A: NULL B: &Head, C: &Tail
    NULL; // Initialiserung-> A: NULL B: &Head, C: NULL
    //Leere Daten:
    ....
};
```

```
// Doppelt verkettete Liste mit separatem Endelement
// oder Abschluss-Sentinel
List Tail = { // Variante C
    NULL; // kein Nachfolger oder &Tail
    NULL; // mit &Head initialisieren!
    //Leere Daten:
    ....
};

List *NewElement(List *Pred)
{
    List *New;

    if( (New = (List *)malloc(sizeof(List))) == NULL )
        return (NULL);

    // Initialisierung Liste
    New->succ = Pred->succ;
    New->pred = Pred;

    Pred->succ = New;
    New->succ->pred = New;

    // Daten-Init; kann auch ausgelassen werden
    ....

    return (New);
}

void RemoveElement(List *Pred)
{
    List *Del;

    Del = Pred->succ;
    if ( Del == NULL )
        return;

    // Liste neu verketten:
    Pred->succ = Del->succ;

    if( Del->succ != NULL )
        Del->succ->pred = Pred;

    free(Del);

    return;
```

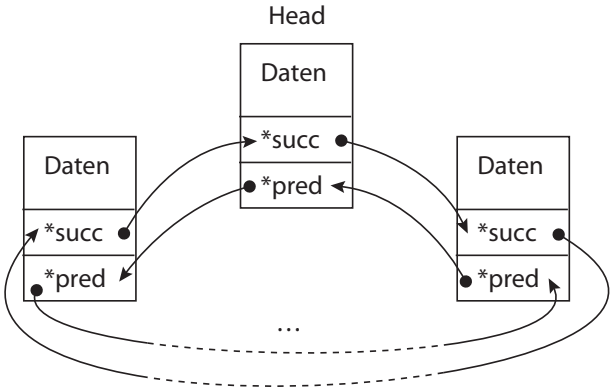



**Abbildung 4.3.:** Doppelt verkettete Liste (Variante B)

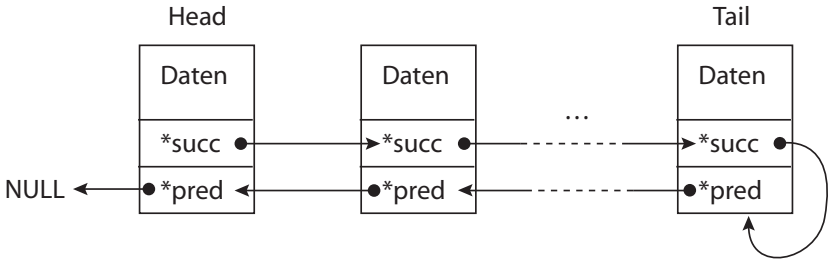



**Abbildung 4.4.:** Doppelt verkettete Liste (Variante C)

```
}

....
```

Zusätzlich kann man am Ende statt einem `NULL`-Zeiger ein Abschlusselement einsetzen, das auf sich selbst zeigt. Verkettete Listen werden überall dort eingesetzt, wo man Arrays braucht, an die man zur Laufzeit noch etwas anhängen kann. Zugriffe auf beliebige Elemente dauern sehr lange, weil man sich im Mittel durch die halbe Liste durchhangeln muss, um zum gewünschten Element zu kommen. Demgegenüber hat der Zugriff auf ein bestimmtes Element eines Feldes einen konstanten geringeren Zeitaufwand. Zum Umsortieren schreibt man die Zeiger auf die Listenelemente in ein Array, sortiert dann nur dieses Array neu und baut die Liste anhand des sortierten Arrays neu auf. Alternativ kann für nur wenige verschiedene Sortierkriterien auch jeweils ein Satz Zeiger mitgeführt werden. Allerdings stellt sich schon beim Sprung von der verketteten Liste auf die doppelt verkettete Liste die Frage, wieviel Aufwand in die Struktur gehen sollte und wieviel in den Inhalt.

Ein *Stapel* ist wesentlich einfacher aufgebaut als eine Liste; im Wesentlichen werden nur Objekte auf den Stapel gelegt und wieder heruntergenommen. Einsatzbeispiel: Wenn ein Funktionsaufruf erfolgt, könnte man alle aktuellen Variablen auf den Stapel legen sowie die Adresse, zu der zurückgesprungen werden soll, dann alle Parameter auf den Stapel und dann zu der Speicheradresse springen, wo die Funktion beginnt. Die Funktion nimmt sich Werte vom Stapel, interpretiert diese als die gewuenschten Parameter, wird ausgeführt, holt sich die Rücksprungadresse und legt schließlich den Rückgabewert auf den Stapel. Dann wird zur aufrufenden Adresse zurückgesprungen. Man nimmt den Rückgabewert vom Stapel, dann werden alle Variablen vom

Stapel geholt und es geht weiter. Ein anderes Beispiel ist der Arbeitsstapel auf einem typischen Schreibtisch... Eine Implementierung kann mittels einer Liste erfolgen, der Eintrag direkt nach dem Kopf ist dann das aktuelle Ende der Liste, das Abschlusselement sozusagen der Boden. Die Einfüge-Aktion nach dem Kopf wird oft mit *push* bezeichnet, das vom Stapel Holen mit *pop*.

Mit Hilfe eines Stapels kann man leicht einen Taschenrechner für die sogenannte umgekehrte polnische Notation programmieren: Die Operation steht immer nach den Operanden, die nur noch vom Stapel geholt werden müssen. Die Eingabe von $3+4$ würde als `3 4 +` erfolgen, bei $(3+4*8)/7$ ergäbe sich $348*+7/$.[11]. In ***C***-Code:

```
.... /* "3" einlesen */ ....
push(3);
.... /* "4" einlesen */ ....
push(4);
.... /* "+" einlesen */ ....
push(pop() + pop();
/* Auf dem Stapel liegt jetzt eine 7 */
....
printf("Ergebnis: %g\n", (double) pop());
```

bzw.

```
push(3); .... push(4); .... push(8); ....
push(pop() * pop()); /* Stapel: 3, 4*8=32 */
....
push(pop() + pop()); /* Stapel: 3+32=35*/
.... push(7); /* Stapel: 35, 7*/
/* push(pop()/pop()) waere falsch */
divisor = pop(); /* Stapel: 35 */
 push(pop()/divisor);  /* Stapel: 5 */
printf("Ergebnis: %g\n", (double) pop());
```

Neben dem fehlenden Test wider Division durch Null ist `push(pop()/pop())` deswegen falsch, weil in ***C*** die Auswertungsreihenfolge von bestimmten Ausdrücken nicht festgelegt ist. Das ist bei den arithmetischen Operationen der Fall. Näheres findet sich im Abschnitt 5.11.

Ein Stapel funktioniert nach dem *LIFO*-Prinzip (*last in, first out*). Speziell für Listen gleichzubehandelnder Aufgaben ist das nicht sinnvoll, da unter Umständen die allererste Aufgabe sehr lange liegenbleibt. Hier sind *Schlangen* sinnvoll. Diese arbeiten nach dem *FIFO*-Prinzip (*first in, first out*) und benötigen als Operationen das Hintenanstellen eines neuen Objekts (*put*) und das Entfernen vom vorderen Ende (*get*). Zur Realisierung braucht man zwei Elemente *head* und *tail*, die Anfang und Ende der Schlange markieren.

Sowohl verkettete Listen als auch Stapel als auch Schlangen können auch rein über Felder realisiert werden.

Den besprochenen eindimensionalen Datentypen stehen prinzipiell vor allem *Bäume* gegenüber. Die wichtigsten Eigenschaften und Grundbegriffe werden im Folgenden kurz erläutert, einen

[11] Wegen Punkt vor Strich muss $4*8$ als erstes berechnet werden; das wird dadurch erreicht, dass `*` vor §+§ geschrieben wird.

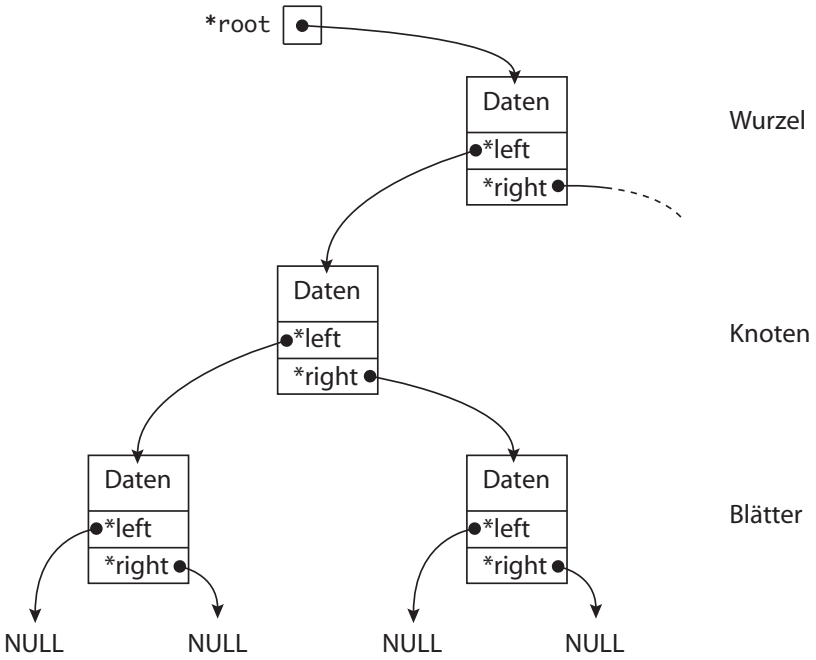



**Abbildung 4.5.:** Baum als Datentyp

Überblick bietet auch Abbildung 4.5: Ein Baum ist eine (nichtleere) Menge von *Knoten* und *Kanten*; eine Kante ist die Verbindung von zwei Knoten. Ein *Pfad* ist eine Liste von Knoten, für die zwischen je zwei aufeinanderfolgenden Knoten eine Kante zwischen diesen beiden Knoten existiert, der Pfad führt vom ersten zum letzten Knoten in der Liste. In einem Baum gibt es genau einen Pfad, der zwei beliebige, paarweise verschiedene Knoten verbindet, d.h. es gibt keine Möglichkeit im Kreis zu laufen; der Baum enthält die minimale Anzahl an Kanten, so dass noch zwischen allen Knoten ein Pfad existiert.

Für die Datenverarbeitung wird ein Knoten des Baumes herausgegriffen und zur *Wurzel* des Baumes erklärt. Die oben angesprochene Eigenschaft ist äquivalent dazu, dass es zwischen der Wurzel und einem anderen Knoten des Baumes genau einen Pfad gibt. In allen anderen Fällen handelt es sich nicht um einen Baum. Man kann die Kanten des Baumes auch als gerichtete Kanten auffassen, die nur von der Wurzel wegführen.

Jeder Knoten kann dann als Wurzel eines Unterbaums gesehen werden, der alle Knoten umfasst, die durch die gerichteten Kanten von ihm aus erreichbar sind. Alle Knoten, die über einen Pfad der Länge $n$ von der Wurzel aus erreichbar sind, liegen auf der $n$-ten Stufe bzw. Ebene. Knoten, die keine Nachfolger bezüglich der gerichteten Kanten haben, heißen *Blätter* oder Endknoten. Die Knoten tragen in der Regel die Informationen. Es ist möglich, dass nicht alle Endknoten bzw. nur Nicht-Endknoten oder nur Blätter Informationen tragen. Solche Situationen können beispielsweise auftreten, wenn gefordert wird, dass alle Nicht-Endknoten genau $n$ Nachfolger haben. Man spricht dann von $n$-ären Bäumen, im einfachsten Fall ($n = 2$) von *binären Bäumen*; die beiden Nachfolger werden dann als linker und rechter Nachfolger bezeichnet.

Binäre Bäume lassen sich einfach mit den bisherigen Mitteln darstellen, indem man jedem Datensatz einen Zeiger/Verweis auf den rechten und linken Nachfolger mitgibt. Zusätzlich braucht man nur noch einen Zeiger auf den Wurzelknoten.

Anwendungs-Beispiele:

- Ein eigener Taschenrechner mit den vier Grundrechenarten: Ein auszuwertender Ausdruck, z.B. $10 * (5 - 3) + 7$, wird so abgespeichert, dass der letzte anzuwendende Operator (hier: $+$) in der Wurzel gespeichert wird und seine Operanden in den Nachfolgern. Sind diese noch keine Zahlen, wird stattdessen wieder ein Baum gebaut, d.h. der rechte Nachfolger wäre 7, der linke ein Unterbaum mit der Wurzel $*$ usw. Die Auswertung des Baums erfolgt dann in ähnlicher Weise: Ist die Wurzel kein Blatt, so werden die Nachfolger der Wurzel ausgewertet und das Ergebnis der in der Wurzel gespeicherten Operation zurückgeliefert. Ist die Wurzel ein Blatt, so wird die gespeicherte Zahl zurückgeliefert. Die Auswertung der Nachfolger bedeutet den Abstieg in den linken und dann in den rechten Teilbaum. Alternativ kann man das Ganze auch über rekursive Funktionsaufrufe regeln: Für die Auswertung ruft man die Funktion `GetAddition()` auf, die nach dem ersten additiven Operator außerhalb von Klammern sucht. Findet sie eins, so ruft sie für den Teil dahinter wieder `GetAddition()` auf und für den Teil davor eine Funktion `GetMultiplikation()` und gibt die Summe bzw. Differenz der Rückgabewerte zurück, ansonsten ruft sie für den Term `GetMultiplikation()` auf und gibt deren Rückgabewert zurück. `GetMultiplikation()` sucht nach dem ersten multiplikativen Operator außerhalb von Klammern, der erste bzw. einzige Teil wird einer Funktion `GetValue()` übergeben, ein etwaiger zweiter Teil wiederum an die Funktion `GetMultiplikation()`, Rückgabe ist wiederum ein Produkt bzw. ein Quotient oder der Rückgabewert von `GetValue()`. Die Funktion `GetValue()` ruft für geklammerte Terme `GetAddition()` mit entfernten Klammern auf, für ungeklammerte Terme wird der Wert geholt und zurückgegeben. Die Abarbeitungsreihenfolge ist die gleiche, wie wenn man erst den Baum aufbauen würde, aber man spart sich den zusätzlichen Speicheraufwand:

```
GetAddition("10*(5-3)+7")
--> (GetMultiplikation("10*(5-3)")+GetAddition("7"))
    --> (GetMultiplikation("10")*GetMultiplikation("(5-3)"))
        --> (GetValue("10")) --> 10
    -2> (GetValue("(5-3)"))
            --> (GetAddition("5-3"))
                --> (GetMultiplikation("5")-GetAddition("3"))
                    --> (GetValue("5")) --> 5
                    -2> (GetMultiplikation("3"))
                        --> (GetValue("3")) --> 3
                    3
                5-3=2
            2
        2
    10*2=20
    -2> (GetMultiplikation("7"))
         --> (GetValue("7")) --> 7
    7
20+7=27
```

- Abspeichern sortierter Daten für schnellen Zugriff: Hierbei spielt die sogenannte *Traversierung* des Baumes eine Rolle. Zum einen gibt es die Level-Order-Traversierung, bei der zuerst die Wurzel, dann alle Knoten der ersten Ebene, dann alle Knoten der zweiten Ebene usw. betrachtet werden, zum anderen gibt es die der Baumstruktur eher angepassten Traversierungsmöglichkeiten pre-order (Wurzel-linker Unterbaum-rechter Unterbaum), in-order

(linker U., Wurzel, rechter U.) und post-order (linker U., rechter U.,Wurzel). Der obige mathematische Ausdruck wird in dieser Darstellung in-order durchlaufen.

Zurück zum schnellen Zugriff, etwa bei der In-Order-Traversierung: Wir suchen ein Datum mit einem bestimmten Schlüssel (z.B. „Hans Müller“). Zunächst vergleichen wir den Schlüssel mit dem der Wurzel. Sind sie gleich, sind wir fertig, ist der gesuchte Schlüssel kleiner, steigen wir in den linken, ist er größer, in den rechten Teilbaum ab und machen dort das gleiche. Ist der Baum ausbalanciert, d.h. sind alle Ebenen bis auf die höchste voll besetzt, hat man bei $N$ Datensätzen nur $n = \lceil \log_2(N+1) \rceil$ Ebenen, also auch nur höchstens $n$ Schritte bis zum Finden des gewünschten Datums. Ist der Baum entartet und man hat z.B. nur linke Nachfolger, dann braucht man bis zu $N$ Schritte; die Schwierigkeit liegt also hier darin, dass von Zeit zu Zeit der Baum ausbalanciert werden muss.

- Abbildung rekursiver und hierarchischer Strukturen: Ein Baum kann auch rekursiv definiert definiert werden (Ein Knoten ist entweder ein Blatt oder hat Nachfolger, die ihrerseits Knoten sind; der Knoten, der kein Nachfolger ist, heißt Wurzel.) und eignet sich dementsprechend auch für die Abbildung von rekursiven Strukturen, etwa Stammbäumen oder Turnierplänen. Die Definition von Ebenen kann zur Abbildung hierarchischer Strukturen, etwa in Unternehmen, herangezogen werden.

# 5. Operatoren

Die eigentliche Arbeit in jedem Programm wird durch die Anwendung von Operatoren in Verbindung mit den Befehlen verrichtet. Operatoren haben eine bestimmte Priorität; Operatoren mit größerer Priorität werden früher angewandt (z.B. „Punkt vor Strich“). Bei Operatoren mit gleicher Priorität spielt die Ausführungsreihenfolge keine Rolle; in der Regel wird hier von links nach rechts gearbeitet (implementierungsabhängig). Die Prioritäten finden sich im Referenzblatt. Die meisten Operatoren sind binäre Operatoren (haben also zwei Operanden). Es gibt aber auch unäre Operatoren sowie einen ternären Operator.

In Tabelle 5.1 sind die Operatoren ihrer Priorität nach aufgeführt. Assoziativität bedeutet hier logische Auswertungsreihenfolge der Operanden, links bedeutet also Auswertung zuerst des linken, dann des rechten Operanden bei binären Operatoren. Logische Auswertungsreihenfolge heißt, dass damit das Ergebnis der Operationen festliegt, aber nicht, dass die Operanden tatsächlich in dieser Reihenfolge ausgewertet werden müssen. Einzige Ausnahmen bilden hier das logische UND, das logische ODER, der Komma-Operator und der Conditional-Operator. Im Abschnitt über *sequence points*, 5.11 wird das näher erläutert.

## 5.1. Arithmetische Operatoren

*Rückgabewert:* Ergebnis der Rechenoperation.

| | |
|---|---|
| $-$ | unäres Minus (Vorzeichen). |
| $+, -, *, /$ | Grundrechenarten. |
| % | Modulo-Operator: Rest der Ganzzahldivision durch den zweiten Operanden. |
| $--$, $++$ | Dekrement bzw. Inkrement um 1, siehe unten; unär. |

## 5.2. Vergleichsoperatoren und logische Operatoren

*Rückgabewert:* Wahr (ungleich Null) oder falsch (Null).

Vergleiche:

| | |
|---|---|
| $==$ | Gleich? |
| $!=$ | Ungleich? |
| $<$, $>$, $<=$, $>=$ | Kleiner, größer, kleiner oder gleich, größer oder gleich? |

Logische Operationen auf Wahrheitswerten:

| | |
|---|---|
| && | UND: Beides wahr? |
| \|\| | ODER: Eins von beiden wahr? |
| ! | NICHT: Aussage falsch? Unär. |

Nach dem ersten Argument von `&&` und `||` liegt ein sequence point (siehe Abschnitt 5.11).

## 5.3. Bitweise Operatoren

*Eingabe:* Ganzzahl-Datentypen.
*Rückgabewert:* Ein Ganzzahl-Datentyp, der das Ergebnis der bitweisen Anwendung des jeweiligen Operators enthält.

| Prio | Operator | Kurzbeschreibung | Assoziativität |
|---|---|---|---|
| 15 | `()` | Funktionsaufruf | links |
| | `[]` | Arrayindex | links |
| | `->, .` | Elementzugriff | links |
| 14 | `!, ~` | Negation (logisch, bitweise) | rechts |
| | `++, --` | Inkrement, Dekrement | rechts |
| | `sizeof` | | rechts |
| | `+, -` | Vorzeichen | rechts |
| | (*Typname*) | typecast | rechts |
| | `*, &` | Dereferenzierung, Adresse | rechts |
| 13 | `*, /` | Produkt, Quotient | links |
| | `%` | modulo | links |
| 12 | `+, -` | Summe, Differenz | links |
| 11 | `<<, >>` | bitweiser Shift | links |
| 10 | `<, <=` | Vergleich kleiner (oder gleich) | links |
| | `>, >=` | Vergleich größer (oder gleich) | links |
| 9 | `==, !=` | Gleichheit, Ungleichheit | links |
| 8 | `&` | bitweises AND (UND) | links |
| 7 | `^` | bitweises XOR (Exklusives ODER) | links |
| 6 | `\|` | bitweises OR (ODER) | links |
| 5 | `&&` | logisches AND (UND) | links |
| 4 | `\|\|` | logisches OR (ODER) | links |
| 3 | `?:` | bedingte Auswertung | rechts |
| 2 | `=` | Zuweisung | rechts |
| | `+=, -=, *=` `/=, %=` `&=, ^=, \|=` `<<=, >>=` | kombinierte Zuweisung | rechts |
| 1 | `,` | Komma | links |

**Tabelle 5.1.:** Operatoren: Priorität und Assoziativität. Alle Operatoren mit Priorität 14 sind unär.

`&` UND: Beide Bits 1? Dann Ergebnisbit 1, sonst 0.

`|` ODER: Eines der beiden Bits 1? Dann Ergebnisbit 1, sonst 0.

`^` EXKLUSIVES ODER: Eines der Bits 1, das andere 0? Dann ...

`~` KOMPLEMENT (NICHT): Ist das Bit 0? Dann ... Unär.

`<<` Linksverschiebung: Hänge rechts entsprechend viele Nullbits an; schneide links gegebenenfalls ab.

`>>` Rechtsverschiebung: Schneide rechts entsprechend viele Bits ab, fülle im Fall von vorzeichenlosen Datentypen von links mit Nullen auf. Bei vorzeichenbehafteten Datentypen sind sowohl eingeschobene Nullen (*logischer Rechtsshift*) als auch eingeschobene Einsen (*arithmetischer Rechtsshift*) zum Vorzeichenerhalt) möglich. Was tatsächlich geschieht, ist implementierungsabhängig.

Beispiel (i.d.R. abgeschnitten auf 11 Binärstellen):

| | |
|---:|---:|
| *p* | $01001101101_2$ |
| *q* | $00100111101_2$ |
| `p&q` | $00000101101_2$ |
| `p\|q` | $01101111101_2$ |
| `p^q` | $01101010000_2$ |
| `~p` | $10110010010_2$ |
| `p<<5` | $10110100000_2$ |
| `(uint16_t)(p<<3)` | $0001001101101000_2$ |
| `(uint8_t)(p<<3)` | $01101000_2$ |
| `q>>2` | $00001001111_2$ |

Das Abschneiden von 12 auf 7 Stellen entspräche der Nachschaltung von & $(000001111111)_2$. Da es in ***C*** keine direkte Binärdarstellung gibt, ist die alternative Verwendung von Hexadezimalkonstanten (oder Oktalkonstanten) sinnvoll, hier etwa für die Beschränkung auf 11 Stellen `& 0x7FF` (bzw. `& 03777`). Auf dieser Basis kann man gezielt Bits ausmaskieren (d.h. auf 0 setzen). Mit `|` können bestimmte Bitbereiche (auf 1) gesetzt werden, mit `^` invertiert werden (alle Einsbits im zweiten Operanden werden invertiert) oder mit den Shift-Operatoren `<<`, `>>` gezielt verschoben werden. Eine Linksverschiebung um 1 entspricht (falls die Binärstellen ausreichen) einer Multiplikation mit 2, eine Rechtsverschiebung um 1 entspricht einer Ganzzahl-Division durch 2.

Bitweises Arbeiten macht nur dort Sinn, wo eine massive Speicherplatz- oder Zeitersparnis erzielt werden muss (und kann).

## 5.4. Zuweisungsoperatoren

*Rückgabewert:* Der Wert, der zugewiesen wurde (bzw. im Fall des Postinkrements/-dekrements der Wert vor der Zuweisung).

| | |
|---|---|
| $=$ | Normale Zuweisung. |
| $+=, -=, *=, /=, \%=$ | Ausführung der arithmetischen Operation mit dem Ziel als erstem Operanden, dann Zuweisung des Ergebnisses. |
| $\&=, \mid=, \hat{}=, \sim=, <<=, >>=$ | Ausführung der bitweisen Operation mit dem Ziel als erstem Operanden, dann Zuweisung des Ergebnisses. |
| $++(), --()$ | Präinkrement/-dekrement: Zuweisung des nächsten/ vorherigen logischen Wertes an das Ziel vor der Rückgabe. |
| $()++, ()--$ | Postinkrement/-dekrement: Zuweisung des nächsten/ vorherigen logischen Wertes an das Ziel nach der Rückgabe. |

## 5.5. Conditional-Operator `?:`

Der einzige ternäre Operator in ***C***.
*Syntax: A?B:C*
*Rückgabewert:* Wenn *A* wahr (nicht Null) ist, *B*, sonst *C*.

Nach dem ersten Argument liegt ein sequence point (→5.11).

## 5.6. `sizeof`-Operator

In K& R-***C*** und ***C***89 ein zur Kompilierungszeit ausgewerter Operator; in ***C***99 wegen flexibler Feldgrößen teilweise zur Laufzeit ausgewertet.
*Syntax: sizeof(typ)* oder *sizeof(variable)*
*Rückgabewert:* Größe von *typ* bzw. des Typs, von dem *variable* ist, in Byte

## 5.7. Typecast-Operator

*Syntax: (typ) variable*
*Rückgabewert:* Das Ergebnis der Umwandlung von *variable* in den Typ *typ*

## 5.8. Operatoren beim Umgang mit Zeigern und Adressen; `*`, `[]`, `&`

**Dereferenzierungsoperator (Inhaltsoperator) `*`:**
*Syntax: *zeiger*
*Rückgabewert:* Ein Wert vom Typ, auf den *zeiger* zeigt, an der Adresse, auf die *zeiger* zeigt.

**Indexoperator `[]`:**
*Syntax: feld[elementindex]* oder *zeiger[elementindex]*

*Rückgabewert:* Wert des *elementindex*+1ten Elements des Arrays *feld* bzw. Wert vom Typ *typ*, auf den *zeiger* zeigt, geholt von der Adresse *zeiger+(elementindex*sizeof(typ))*.

**Referenzierungsoperator (Adressoperator) &:**
*Syntax: &variable*
*Rückgabewert:* Adresse, an der *variable* gespeichert ist.

## 5.9. Auswahl-Operatoren `->`, `.`

Element-Auswahl bei den komplexen Datentypen Struktur, Vereinigung und Bitfeld.
*Syntax: kplx.elementname* bzw. *kplxzeiger−>elementname*
Hierbei ist *kplxzeiger−>elementname* äquivalent zu *(*kplxzeiger).elementname*.
*Rückgabewert:* Wert von *elementname*.

## 5.10. Komma-Operator `,`

Aneinanderreihung von Anweisungen, vor allem in der Initialisierung und im Inkrement der `for`-Schleife. Operator mit der niedrigsten Priorität.
*Syntax: ersteAnweisung, zweite Anweisung*.
*Rückgabewert:* Rückgabewert der letzten Anweisung.

Nach dem ersten Argument liegt ein sequence point.

## 5.11. Sequence points

Wie bereits mehrfach erwähnt, liegt bei vielen Operationen die tatsächliche Auswertungsreihenfolge nicht fest, zum Beispiel hier:

```
i = 1;
i = ++i + i++;
```

Kommt hier nun 2, 3 oder 4 raus? Nicht bekannt. Auch ohne die Addition oder die Zuweisung käme nichts eindeutiges heraus.

Woran liegt das? Es gibt keinen *sequence point* in der zweiten Zeile. Sequence points sind ein Kunstgebilde, das eindeutige von uneindeutigen Operationen abgrenzt. Zwischen zwei aufeinanderfolgenden sequence points darf jede Variable höchstens einmal modifiziert werden. Die Reihenfolge, in der dazwischenliegende Ausdrücke ausgewertet werden, liegt nicht fest; Assoziativität und Vorrang der Operatoren legen nur fest, wie das Ergebnis auszusehen hat.

Das ist dort kritisch, wo es auf die Reihenfolge ankommt und man Nebeneffekte von Operationen erwartet, die aber erst zum nächsten sequence point eintreten.

```
/* B() und C() veraendern ihr Argument */
```

```
A = B(&i) + C(&i);
/* Entweder wir kriegen das gewünschte Ergebnis oder das
** additive Inverse: */
push(pop() - pop());
/* Das gibt nicht notwendigerweise 1, 1, 2 aus */
i = 20;
printf("%d, %d, %d\n", i=1, i++, i++);
```

Wo kann man nun mit sequence points rechnen? Nach dem Ende einer Anweisungszeile (mit Semikolon abgeschlossen), einer Deklaration mit Initialisierung, nach dem ersten Operanden des Komma-Operators sowie des Conditional-Operators, von logischem UND und ODER, *nach* einem Funktionsaufruf, d.h. nach der in beliebiger Reihenfolge erfolgten Auswertung der Argumente, nach den Kontrollausdrücken von `if`, `switch`, `while`, `do…while` und jedem der Ausdrücke in einer `for`-Schleife sowie nach dem Ausdruck in einer `return`-Anweisung und an ein paar anderen Stellen (in Verbindung mit Bibliotheksfunktionen).

Was heißt das?

```
if (a<b)
    tmp = a, a = b, b = tmp;
C = (a<b) ? ++a : b++;
if (s != NULL && strcmp(s, "Tulpe") == 0)
    printf("Dolle Sache\n");
if (s == NULL || strcmp(s, "Tulpe"))
    printf("Schade\n");
```

funktionieren, d.h. es findet ein Dreieckstausch statt, `a` bzw. `b` werden erst inkrementiert, wenn klar ist, wie der Vergleich ausgegangen ist, und wir übergeben `strcmp()` in keinem Fall einen Null-Zeiger.

# 6. Funktionen

Das wichtigste über Funktionen ist bereits bekannt; hier soll noch einmal das Bekannte zusammengefasst werden und ein paar Spezialitäten angesprochen werden.

## 6.1. Grundlegendes

Eine Funktion wird nach dem Muster *Rückgabetyp Funktionsname (Parameterliste);* deklariert und durch *Rückgabetyp Funktionsname (Parameterliste) { Funktionsrumpf }* definiert. Die Rückgabetypen können in ***C*** von einem beliebigen elementaren oder komplexen Datentyp sein. Das gleiche gilt für die Parameterliste. Die einzige Ausnahme stellen in beiden Fällen (mehrdimensionale) Felder beliebiger Größe dar, siehe auch 4.7.3.

Die bei der Deklaration (Prototypierung) oder Definition angegebenen *Parameter* sind eine formale Beschreibung der Werte, die an die Funktion übergeben werden. Die Werte, die bei einem Funktionsaufruf übergeben werden, nennt man im Gegensatz dazu *Argumente*. Diese Unterscheidung ist aber im normalen Sprachgebrauch eher unüblich.

Der mögliche Ablauf eines Funktionsaufrufs wurde im Abschnitt über Stapel bereits angedeutet, verkürzt sieht der Weg von Argumenten und Rückgabewert so aus: Alle Argumente werden beim Aufruf auf den Programm-Stapelspeicher (Stack) gepackt, in der Funktion wieder heruntergeholt, verwendet und schließlich kommt der Rückgabewert (wenn nicht `void`) beim Rücksprung an die Aufrufstelle ebenfalls auf den Programm-Stapelspeicher. Dieses Umkopieren über den Stack ist der Grund, warum man Felder oder große komplexe Daten besser über einen Zeiger übergibt als direkt, weil dann nur der Zeiger auf den Stack gelegt und wieder vom Stack heruntergenommen werden muss.
**Hinweis:** Ein Funktionsaufruf muss nicht so ablaufen, allerdings kann man bei vielen Systemen davon ausgehen, dass der Ablauf sich so oder so ähnlich darstellt.

Die reine Deklaration einer Funktion wird auch als Funktions*prototyp* bezeichnet, weil nur die äußere Form beschrieben wird, der Funktion aber beliebige Funktionszeiger zugewiesen werden können. Für einen Funktionsprototypen sind bereits Rückgabetyp und Funktionsname ausreichend. Die Parameterliste ist nicht zwingend erforderlich und kann sich sogar von Prototyp zu Prototyp und in der eigentlichen Definition unterscheiden! Es ist aber sinnvoll, immer die volle und gleiche Parameterliste anzugeben, um dem Compiler das Aufspüren möglicher Fehler zu ermöglichen und um selbst eine Gedächtnisstütze zu haben.

Variablen und Funktionen teilen sich den Raum, aus dem die Namen sein dürfen. Jeder Funktionsname entspricht einem Zeiger auf die entsprechende Funktion.

## 6.2. `main`

In jedem Programm gibt es eine Funktion namens `main`. Sie ist die Stelle, an der das eigentliche Programm startet, nachdem der Initialisierungscode durchgeführt wurde. In der Regel wird das Programm auch durch Erreichen des Endes von `main` bzw. einer `return`-Anweisung in `main` beendet (danach gibt es noch Code, der etwas aufräumt). Mit den Funktionen `exit()`, `abort()` und `atexit()` der Standard-Bibliothek kann ein etwas anderes Verhalten bei der Beendigung erreicht werden.

Für `main` gibt es, im Gegensatz zu allen anderen Funktionen zwei Möglichkeiten für die Parameterliste. Bis jetzt haben wir oft nur die leere Parameterliste betrachtet; man kann aber - wie bei jedem anderen Shell-Kommando auch - einem Programm Kommandozeilen-Parameter mitgeben. Damit diese im Programm ausgelesen werden können, wird `main` anders deklariert:

```
/*******************************************************
**
**  main - Hauptprogramm, das seine Parameter ausgibt
**
**  Argumente: argc - Anzahl der Strings, die in argv
**                    gespeichert sind
**             argv - Stringfeld mit den Parametern;
**                    argv[0] enthaelt i.d.R. den
**                    Programmnamen
**
**  Rueckgabe: 0 .. alles okay
**
*/

int main (int argc, char **argv)
{
    if (argc>0)
        printf("Mein Programm heisst %s und hat"
               "%d Parameter uebergeben bekommen\n",
               argv[0], argc-1);

    for (int i=1; i<argc; i++)
        printf("%d-ter Parameter: %s\n", argv[i]);

    return 0;
}
```

Dass der Wert 0 zurückgegeben wird, wenn alles richtig gelaufen ist, mag im ersten Moment etwas kontraintuitiv wirken, weil 0 ja auch für `false` steht. Es ist aber für Abfragen leicht zu testen, ob eine Funktion eine Rückgabe gleich oder ungleich Null geliefert hat und nur im Fall ungleich nachzuschauen, was es denn für ein Fehler war bzw. wo er aufgetreten ist, da es meistens mehrere Fehlermöglichkeiten und nur einen richtigen Ausgang gibt. Diese Vorgehensweise ist generell zu empfehlen, wenn man keinen Wert als Rückgabe braucht; als Nicht-Null-Rückgabewert ist z.B. die Zeilennummer ganz nützlich, falls man nicht Fehlercodes über symbolische Konstanten

definieren will.

```
if (err = funktion(a, b, c)) {
    printf("Fehler in funktion, Fehlercode %d\n",err);
    return(__LINE__);
}
```

## 6.3. Effiziente Such- und Sortieralgorithmen

### 6.3.1. Beispiel Suche

```
int suche(int *feld,int len,int val)
{
    int i;
    for (i=0;i<len;i++)
    {
        if (feld[i]==val)
            return(i);
    }
}
```

Bemerkung:

- Befindet sich val Mehrfach im Feld,so wird der kleinste Index zurückgegeben
- Für zufällige Werte im Feld muss im Schnitt das halbe Feld durchlaufen werden, bevor der Wert gefunden wurde
- Befindet sich der gesuchte Wert nicht im Feld, so muss das gesamte Feld durchlaufen werden!
- Leider die einzige (und schnellste) Möglichkeit für beliebige Felder

Verbesserter Algorithmus: Unter der Annahme, dass das Feld sortiert ist.

Idee: Divide and Conquer-Strategie (Teile und herrsche)

### 6.3.2. Binary Search (Binäre Suche)

- Vergleiche mit dem mittleren Wert
- Falls gleich: gefunden
- Sonst: Suche in entsprechenden (halb so großen) Teilfeld

```
int binaere_suche(int *f,int start, int ende, int wert)
{
    int pos;

    if (ende<start) //Feld leer
        return(-1);

    if (ende==start)    //1 Element im Feld
        if (f[ende]==wert)
            return(ende);
        else
            return(-1);

    pos=(start+ende)/2;
    if (f[pos]==wert)
        return(pos);
    else if (f[pos]>wert)
        return binaere_suche(f,start,pos-1,wert);
    else
        return binaere_suche(f,pos+1,ende,wert);
}
```

Bemerkungen:

- Der Aufwand beträgt jetzt O(log(n))

## 6.4. Rekursion

Manche Probleme und Strukturen sind von Haus aus rekursiv definiert. In solchen Fällen kann es sinnvoll sein, Funktionen, die auf ihnen operieren, rekursiv zu definieren, d.h. in einer Weise, dass sich die Funktionen selbst aufrufen. Die Berechnung von *n*! beispielsweise könnte so definiert sein:

```
// Berechne n!
long Factorial(long n)
{
    return (n ? n*Factorial(n-1) : 1);
}
```

In diesem Fall ist die Verwendung einer Rekursion nicht notwendig - und auch nicht unbedingt sinnvoll, da auf diese Weise bestenfalls die Sprungvorhersage des Prozessors im Vergleich zu Schleifen ausgebremst wird. Außerdem wird das Abfangen von Fehlern gleich sehr aufwendig, da sozusagen eine Fehlerabfrage (z.B. ob *n* nichtnegativ ist) bei jedem Aufruf wieder gemacht wird, während das ganze für eine Funktion mit einer Schleife durch eine der Schleife vorgelagerte Abfrage erledigt werden kann.

Rekursive Funktionen machen immer dann Sinn, wenn

- sie keine zusätzlichen Redundanz in die Problembehandlung einbringen;
- eine Problem-Aufteilung z.B. in kleinere Teilprobleme stattfindet;
- durch die Problemstruktur wenig über die Rekursionstiefe und das Endergebnis vorhergesagt werden kann;
- ein Algorithmus aus diesem oder anderen Gründen nicht anders implementierbar ist.

Insbesondere Algorithmen, die auf Bäumen arbeiten oder Probleme baumartig in Teilprobleme zerlegen, erfüllen diese Bedingung; symbolisch aufgeschrieben könnte ein solcher Algorithmus für eine binäre Baumstruktur mit In-Order-Problembehandlung etwa so aussehen:

***LöseProblem (****Teilproblem****)***
***Wenn*** *Teilproblem* ***klein genug: Gib*** *Lösung* ***zurück.***
***Sonst: Identifiziere*** *Teilproblemwurzel* ***und*** *linkes* ***und*** *rechtes Teilproblem****,***
*LösungLinks =* ***LöseProblem (****linkes Teilproblem****),***
***Verknüpfe*** *LösungLinks* ***mit*** *Lösung* ***für*** *Teilproblemwurzel****,***
*LösungRechts =* ***LöseProblem (****rechtes Teilproblem****),***
***Verknüpfe bisherige*** *Lösung* ***mit*** *LösungRechts****,***
***Gib*** *Lösung* ***zurück.***

Ein Beispiel für eine reale Anwendung (beim Parsen der Grundrechenarten) findet sich im Abschnitt über Bäume auf Seite 94. Ein Nachteil rekursiver Funktionen ist der erhöhte Speicherbedarf beim Ablegen der Argumente auf dem Stack.

Beispiel (Springer-Schachbrett-Problem):

```
void springer(int i,int j,int num)
{
    if(Valid_Index(i,j)&&num<FELD(i,j))
    {
        FELD(i,j)=num;
        springer(i+2,j+1,num+1);
        springer(i-2,j+1,num+1);
        springer(i+1,j+2,num+1);
        springer(i-1,j+2,num+1);
        springer(i+2,j-1,num+1);
        springer(i-2,j-1,num+1);
        springer(i+1,j-2,num+1);
        springer(i-1,j-2,num+1);
    }
}

int main(void)
{
    //Aufruf und Initaialisierung
    //Definiere Schachfeld in vorgegebener Größe
```

```
    int FELD[64],i,j;

    for(i=0;i<8;i++)
        for(j=0;j<8;j++)
            FELD(i,j)=1000;
    springer(2,5,0);
}
```

### 6.4.1. Sortieren

#### Wie ein Stapel Papier

```
void sort(int *f,int len)
{
    int i,j,value,pos;

    for (i=1;i<len;i++)
    {
        pos = i;
        value = f[i];
        // Suche Eintrag links von i, der kleiner als value ist
        while (f[pos-1]>value && pos>0)
            pos--;
        // Füge den aktuellen Wert ein
        for (j=i;j>pos;j--)
            f[j]=f[j-1];
        f[pos] = value;
    }
}
```

Bemerkungen:

- Laufzeit hängt vom Zustand des Feldes ab
- Sortiert in O(n)
- absteigend sortiert: 1+2+3+... = $O(n^2)$

Idee: Zerlege Feld in 2 Teilfolgen

#### Partitionierung

```
int partitioniere (double *f,int von, int nach)
{
    double p=f[von]; //Pivoelement
    int act=von;
```

```
    int i;
    for (i=von+1;i<=nach;i++)
    {
        if (f[i]<p)
        {
            act++;
            if (i!=act)
                tausche(f,i,act);
        }
    }
    tausche(f,von,act);
    return(act);
}
```

Bemerkungen:

- p nennt man auch Pivot-Element
- Aufwand vom Partitionieren: O(n)

#### Quicksort

```
void quicksort{double *f,int von, int nach)
{
    int pivot_pos;
    if (von<=nach)
        return;
    pivot_pos = partitioniere(f,von,nach);
    quicksort(f,von,pivot_pos-1);
    quicksort(f,pivot_pos+1,nach);
}
```

Bemerkungen:

- Aufwand hängt stark von der Partitionierung ab.
- Idealfall: Feld zerfällt in 2 gleich große Teile ( O(log(n)) rekursive Aufrufe, Gesamtaufwand: $O(n * log(n))$ )
- Worst-Case: 1 Teilfeld mit 1 Element ( O(n) rekursive Aufrufe , Gesamtaufwand: $O(n^2)$ )
- Bei Pivotisierung nach dem 1.Element tritt bei sortierten Feldern der worst-case auf.
- Ausweg: Pivotisierung nach mittlerem / zufälligem Element

Siehe auch Katpiel 9.5.3

## 6.5. `inline`-Funktionen

Wenn ein Code-Schnipsel öfter vorkommt, ersetzt man ihn durch eine Funktion. Wenn diese Funktion eigentlich recht kurz ist, überlegt man sich oft, ob man sie nicht durch ein Makro realisiert, das dann u.U. über mehrere Zeilen geht. Das ist aber auch nicht unbedingt übersichtlich.

Durch ***C***++ ist die Idee der *inline*-Funktionen mit ***C***99 in den ***C***-Sprachumfang aufgenommen worden: Die Funktion wird als Funktion definiert, aber mit dem zusätzlichen Schlüsselwort `inline` versehen, das dafür sorgt, dass die Funktion in der Regel direkt an Ort und Stelle (ohne echten Funktionsaufruf) eingebaut wird, sich in dieser Hinsicht also ähnlich wie ein Makro verhält. Wie bei dem Schlüsselwort `register` wird aber `inline` sinngemäß als „verwende die effizienteste Implementierungsvariante“ aufgefasst und die letzte Entscheidung dem Compiler überlassen.

```
#include <stdio.h>

inline int imax(int a, int b)
{
    return  ( (a>b) ? a : b) );
}

int main ()
{
    int erste=5, zweite=3;

    printf("Maximum von %d und %d ist %d",
           erste, zweite, imax(erste, zweite) );

    return (0);
}
```

Der Unterschied zu Makros ist, dass die Datentypen immer noch richtig verwendet werden, man also eine inline-Funktion nicht versehentlich auf Parameter vom falschen Typ anwenden kann.

## 6.6. Variable Argumentlisten

Funktionen wie `printf` sind im Gegensatz zu den Funktionen, die wir bisher verwendet haben, mit einer beliebigen Anzahl an Argumenten aufrufbar. Der „Trick“ besteht bei `printf` darin, dass das erste Argument, der Formatstring, fest ist. Die Funktion wertet dann typischerweise den Formatstring aus und findet heraus, wieviele Argumente von welchem Typ erwartet werden und wie lange der Zielstring höchstens wird und wendet sich dann erst dem variablen Teil zu. Hier werden dann der Reihe nach die erwarteten Argumente aus einer Liste bzw. vom Stack geholt und in den Ziel-String eingebaut. Von der Implementierung her sieht das für eine Summation beliebiger Summandenzahlen folgendermaßen aus:

```
#include <stdarg.h>

double N_Summe (int N, ...)
```

```
{
    double summe=0.0;
    va_list arglist; // Variable-Argumente-Liste

    /* Initialisierung von arglist; zeigt auf das
    ** erste Element nach N.
    */
    va_start(arglist,N);

    /* Argumentliste durchgehen, va_arg liefert
    ** Argument vom gewuenschten Typ und schaltet
    ** zum naechsten Element weiter.
    */
    for (int i=0; i<N; i++)
        summe += va_arg(arglist, double);

    /* Speicher fuer Argumentliste freigeben und
    ** aufraeumen.
    */
    va_end(arglist);

    return summe;
}

int main (void)
{
    double s;

    s = N_Summe(5,20.0,1.0/3.0,0.57,1e-4,0.25);

    printf("Summe: %f.\n", s);

    return 0;
}
```

Mindestens ein Argument der Funktion mit variabler Argumentzahl muss fest sein, um sicherstellen zu können, dass die Argumentzahl bekannt ist. Das Vergessen des abschließenden `va_end()`-Aufrufes führt unter Umständen zu nicht freigegebenem Speicher, man schaufelt sich bei häufigem Aufruf (etwa bei Rekursionen) den Speicher dauerhaft voll.

Es ist auch möglich, einer Funktion ein Argument vom Typ `va_list` zu übergeben; das ist dann nützlich, wenn eine Funktion mit variabler Argumentliste einer anderen Funktion einen Teil oder alle ihre Argumente übergeben will. Will man beispielsweise eine Funktion schreiben, die genau wie `snprintf()` in einen String schreibt, aber den Speicher für diesen String vorher alloziert, dann muss man lediglich die Allozierung des Zielstrings übernehmen und kann den Zielstring, den Formatstring und den variablen Teil der Argumente einfach an `vsnprintf()` (siehe auch Abschnitt C.2) übergeben.

**Hinweis:** Die Anzahl der Argumente muss zwar nicht bei der Funktionsdefinition bekannt sein, aber beim Kompilieren des Programms. Eine beliebige Anzahl von Argumenten zur Laufzeit, indem man sich selbst eine `va_list` bastelt, ist nicht möglich.

## 6.7. Felder in der Parameterliste

Felder können durch Zeiger übergeben werden. Weiß man die Größe eines Arrays im Voraus, kann man diese aber auch angeben; sie ist dann fest. Beim Übergeben mehrdimensionaler Arrays müssen alle Feld-Dimensionen bis auf die letzte fest sein.

Unter ***C***99 ist die Bedeutung von `static` erweitert worden: Man kann mit `static` in einem Array-Parameter mit Größe angeben, dass das Array mindestens von einer bestimmten Größe ist. Das ist nur für die letzte Feld-Dimension möglich (sonst wäre keine Berechnung möglich.

Nachdem es möglich ist, auch Felder variabler Größe zu übergeben, gibt es einen Mechanismus, der es ermöglicht, diese variable Größe auch mitzuteilen. Für Prototypen kann das auch mit einem `*` geschehen.

```
#define DIM0 3
#define DIM1 2

int f1 (size_t n, int *a);  // Groesse und Anfangsadresse
int f2 (size_t n, int a[]); // dito
int f3 (int a[DIM0]);       // Groesse ist bekannt
int f4 (int a[DIM1][DIM0]); // dito, zweidimensional
int f5 (size_t n, int a[][DIM0]); // Feld von int-Feldern
                            // der Groesse DIM0
int f6 (size_t n, int (*a)[DIM0]); // dito
int f7 (size_t n, int *a[DIM0]); // Feld von Zeigern auf
                            // int der Groesse DIM0
int f8 (int a[static DIM1][DIM0]); // C99: Feld von
                          // int-Feldern der Groesse DIM0
                          // mit mindestens DIM1 Elementen
int g1 (int n, int m, double b[n][m]); // C99: flex. Feld
int g2 (int n, int m, double b[*][*]); // C99: flex. Feld
                                       // (nur Prototyp)
int g3 (int n, int m, double b[][*]);  // C99: flex. Feld
                                       // (nur Prototyp)
int g4 (int n, int m, double b[][m]);  // C99: flex. Feld
```

## 6.8. Module und Bibliotheken

Wenn man mehrere Funktionen für einen Zweck oder den Umgang mit einem bestimmten Datentyp geschrieben hat, ist es sinnvoll, sie in einer eigenen Datei zu speichern und über eine Header-Datei in der aufrufenden Datei zugänglich zu machen. Die einzelnen Dateien werden auch als

Module bezeichnet. Hat man beispielsweise Module für den Umgang mit speziellen Strukturen, für das Sortieren und das Suchen auf ihnen u.ä., so können diese, wenn sie allgemein genug gehalten sind, immer wieder verwendet werden. Andererseits ist es auch bei spezialisierten Funktionen sinnvoll, sie in ein eigenes Modul „wegzupacken“, weil dadurch die Übersichtlichkeit an anderer Stelle gewahrt wird. Wenn man nach den wichtigen Funktionen jedesmal zwischen einem Wust von unwichtigen Funktionen suchen muss, kann man sehr viel Zeit verlieren.

Außerdem benötigt man oftmals von „außen“ gesehen jeweils nur einen Teil der Funktionen, so dass auch nur dieser Teil zugänglich ist. Irgendwelche Hilfsfunktionen sind dann nur innerhalb des jeweiligen Moduls sichtbar.

Hat man eine Sammlung nützlicher Module, die man eigentlich immer wieder verwendet, kann man diese zu einer Bibliothek zusammenfassen. Diese enthält gewissermaßen die Module in vorkompilierter Form; man braucht nach außen hin nur die Bibliotheks-Header und die eigentliche Bibliothek. Bibliotheken können statisch oder dynamisch gelinkt werden. Statisch heißt, dass effektiv das gleiche passiert, als wenn man die Module, aus denen die Bibliothek besteht, dem Programm hinzufügt. Dynamisches Linken sagt dem Programm dagegen nur, dass es sich zur Laufzeit an die entsprechende Bibliothek wenden soll, wenn es eine Funktion aus ihr braucht. Im Gegenzug bleibt dafür die Programmdatei kleiner.

Das dynamische Linken hat in einem System, in dem viele Programme die gleichen Bibliotheken verwenden, den Vorteil, dass weniger Speicherplatz verschwendet wird. Der Nachteil ist, dass man die Bibliothek mitnehmen muss, wenn man das Programm auf einem anderen Rechner laufen lassen will. Außerdem kann es passieren, dass unterschiedliche Bibliotheksversionen bei schlechter Programmierung unterschiedlich reagieren und deswegen bestimmte Programme nur mit bestimmten Bibliotheksversionen zusammenarbeiten.

Das statische Linken großer Bibliotheken hat den Nachteil, dass oft nur ein kleiner Teil der Bibliotheksfunktionen gebraucht wird, aber je nach System der Ballast beim Linken bleibt, man also unnötig große Programmdateien bekommt.

Wie man eigene Bibliotheken erstellt, wird im Abschnitt 8.4 behandelt.

# Teil II.

# Kür

## Übersicht

*Nach der Pflicht…*

*Ohne Ein- und Ausgabe nützen die schönsten Programme nichts, daher widmen wir uns etwas ausführlicher den Funktionen aus* `<stdio.h>`*, nachdem die Sprache jetzt bekannt ist.*

*Danach geht es weiter mit Werkzeugen, die beim Programmieren (nicht nur in* ***C****) nützlich sind; hier wird nur eine bei weitem nicht erschöpfende Auswahl vorgestellt. Wenn der eine oder die andere sich etwa in Anbetracht von splint denkt, „Na prima, das hätte ich am Anfang des Kurses viel besser brauchen können!“, sei hier kurz gesagt, dass man viele Fehler erst versteht, wenn man sie selbst auf die harte Tour erlebt hat.*

*Zum Schluss widmen wir uns einem Thema, zu dem die beiden Grundregeln lauten:*

1. *Don’t do it.*
2. *(For experts only): Don’t do it yet.*

*Richtig, es geht um Optimierung. Zuerst sehen wir uns die Mikrooptimierung an, die heute kaum noch eine Rolle spielt, auf die die beiden Regeln zutreffen – und an die immer noch viel Zeit verschwendet wird. Nichtsdestotrotz* kann *es Fälle geben, in denen sie nützlich ist. Abschließend sehen wir uns an, wie man durch bessere Algorithmen optimieren und wie man Algorithmen verbessern kann.*

# 7. Datei-Ein- und Ausgabe

## 7.1. Alles ist eine Datei

***C*** ist für UNIX-Betriebssysteme entwickelt worden. Deshalb orientiert sich das Ein- und Ausgabekonzept von ***C*** an den UNIX-spezifischen Gegebenheiten. Insbesondere gibt es für Tastatureingaben und Bildschirmausgaben die „Dateien“ (oder besser: Datenströme) `stdin` und `stdout`, sowie für Fehlerausgaben den Strom `stderr`.

Auch logische Geräte wie Festplatten oder CDs sind zunächst einmal als Datei eingebunden, deren Inhalt in einem logischen Gerät mit Hilfe geeigneter Treiber sichtbar wird und seinerseits als Ansammlung von Verzeichnissen und Dateien interpretiert wird. Verzeichnisse ihrerseits sind eine spezielle Art von Dateien...

Dateien können in verschiedenen Modi geöffnet werden, z.B. nur zum Lesen, nur zum Schreiben, gemischt. Außerdem können sie auf verschiedene Weise geöffnet werden. Für UNIX bzw. den POSIX-Standard gibt es Low-Level-Funktionen, die betriebssystemnah byteweises Arbeiten mit einer Datei ermöglichen und die mittlerweile oft auch unter anderen Betriebssystemen zur Verfügung stehen, wenngleich sie nicht zum Sprachstandard gehören. ***C*** selbst bietet High-Level-Funktionen, die bestimmte Dinge bereits automatisch machen. Beispielsweise ist nicht festgelegt, welche Sequenz von Zeichen einen Zeilenumbruch bedeutet; die high-level-Funktionen wandeln das ***C***-Zeichen `'\n'` in das/die entsprechende(n) Steuerzeichen für das Betriebssystem um und umgekehrt. Außerdem ist es möglich, dass das Betriebssystem noch einen eigenen Satz an Funktionen für den Umgang mit Dateien mitliefert.

Wir werden uns im Folgenden auf die High-Level-Dateifunktionen von ***C*** beschränken.

## 7.2. Öffnen und Schließen

Der Zugriff auf Dateien erfolgt über die Struktur `FILE`, die beim Öffnen initialisiert wird. In dieser Struktur steht die Information, wo sich die Datei befindet (im sogenannten *Datei-Deskriptor/Handle*), wo die Datei anfängt, wo man sich in der Datei befindet (ganzzahliger *offset* oder Zeiger oder komplizierteres...), ob das Dateiende erreicht wurde, ob Fehler aufgetreten sind (beides in den *Flags*) und ähnliches. Die Funktionen, die mit Dateien umgehen, erwarten in der Regel einen Zeiger auf diese Struktur.

```
#include <stdio.h>

FILE *fopen(const char *path, const char *mode);
FILE *freopen(const char *path, const char *mode,
              FILE *stream);
```

| Modus | Beschreibung |
|---|---|
| `r` | Öffnet Datei nur zum Lesen |
| `w` | Öffnet Datei nur zum Schreiben; erzeugt ggf. eine neue Datei oder überschreibt eine bereits vorhandene |
| `a` | Öffnet Datei zum Anfügen; erzeugt ggf. eine neue Datei oder öffnet die alte und setzt die momentane Position auf das Dateiende, so dass neue Daten angefügt werden |
| `r+` | Öffnet Datei zum Lesen und Schreiben; die momentane Position ist der Dateianfang, die Datei muss bereits existieren. |
| `w+` | Öffnet Datei zum Schreiben und Wieder-Lesen; erzeugt ggf eine neue Datei oder überschreibt eine bereits vorhandene. |
| `a+` | Öffnet Datei zum Anfügen und Wieder-Lesen; wie `a` plus Lesemöglichkeit. |

**Tabelle 7.1.:** Dateizugriffs-Modi für die Verwendung mit `fopen` und `freopen`. Zusätzlich kann der Modus noch durch `b` ergänzt werden, z.B. `r+b`; damit ist man im Binärmodus.

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

Mit `fopen()` wird die Datei, die mit dem String `path` beschrieben wird, geöffnet. `path` kann hierbei der Dateiname oder der Dateiname mit vorangestelltem absolutem oder relativem Pfad sein. Bei Erfolg des Öffnens wird ein `FILE *` zurückgegeben, der „auf die Datei zeigt“, bei Misserfolg `NULL`. Die Datei wird hierbei in einem bestimmten Zugriffs-Modus geöffnet, der regelt, welche Operationen mit der Datei möglich sind, siehe auch Tabelle 7.1. Dieser Modus steht in einem String. Wie oben bereits angesprochen, wird die betriebssystemeigene Zeichenfolge für `'\n'` automatisch umgewandelt. Will man nicht Textdateien, sondern Binärdateien bearbeiten, muss man zusätzlich an den Modus ein `b` anhängen. Unter UNIX gibt es keinen Unterschied.

Die Funktion `fclose()` dient zum Schließen einer Datei und gibt im Erfolgsfall Null zurück.

Mit `freopen()` kann man die Folge aus `fclose(stream)` und anschließendem `stream=fopen(....)` durch einen einzigen Aufruf ersetzen. Die Rückgabe ist wie bei `fopen()`.

Durch Einbinden von `<stdio.h>` stehen auch die drei Dateien `stdin` (nur lesen), `stdout` (nur schreiben) und `stderr` (nur schreiben) zur Verfügung. Unter UNIX gibt es die Möglichkeit, Dateiströme umzuleiten; daher ist es sinnvoll, Fehlermeldungen und normale Ausgaben zu trennen, so dass man die Ausgabe etwa in eine Datei umlenken kann, Fehlermeldungen aber nach wie vor auf der Konsole erscheinen.

## 7.3. Lesen und Schreiben

Ein- und Ausgabe kann auf verschiedene Arten stattfinden, nämlich

- byteweise/zeichenweise,
- stringweise,
- formatiert und
- blockweise (binär).

### 7.3.1. Zeichenweises Lesen und Schreiben

```
#include <stdio.h>

int fgetc(FILE *stream);
int getc(FILE *stream);
int getchar(void);
int ungetc(int c, FILE *stream);
int fputc(int c, FILE *stream);
int putc(int c, FILE *stream);
int putchar(int c);
```

Mittels `fgetc()` oder dem äquivalenten `getc()` kann man ein einzelnes Zeichen aus der Datei auslesen. Die Konsolenfunktion `getchar()` ist realisiert durch `getc(stdin)`.

Manchmal muss man ein Zeichen mehr lesen, als man eigentlich haben will, etwa, um festzustellen, ob das Einlesen eines Strings/Objektes bereits beendet ist, oder man stellt fest, dass man das eingelesene Zeichen aus anderen Gründen nicht brauchen kann. Dann ist es mittels `ungetc()` möglich, *genau ein* Zeichen wieder „zurückzuschicken“. Manche Implementierungen erlauben u.U. auch mehr als einen solchen „Undo“-Vorgang.

Soll ein Zeichen in eine Datei geschrieben werden, kann man das mittels `fputc()` oder dem äquivalenten `putc()` tun. Die Funktion `putchar()` ist realisiert durch `putc(stdout)`.

### 7.3.2. Stringweises Lesen und Schreiben

```
#include <stdio.h>

char *fgets(char *s, int size, FILE *stream);
char *gets(char *s);
int fputs(const char *s, FILE *stream);
int puts(const char *s);
```

Beim stringweisen Einlesen ist eine natürliche Schranke durch die Größe des Strings gegeben. Deswegen liest man mittels `fgets` eine Zeile oder aber höchstens `size-1` Zeichen ein. Das Zeichen `'\n'` wird, wenn es gefunden wurde, mit in den String hineingeschrieben, der Stringterminator wird als letztes Zeichen angehängt. Im Fehlerfall oder am Dateiende wird `NULL` zurückgegeben, sonst ein Zeiger auf den Stringanfang. Die Funktion `gets()` liest von `stdin`, aber

ohne Beschränkung der Eingabelänge, weswegen ihre Anwendung unsicher und fehlerträchtig ist[1].

Mittels `fputs()` kann man einen String in eine Datei schreiben. Bei Erfolg wird ein positiver Wert zurückgeliefert, sonst der Wert `EOF` (symbolische Konstante für *end of file*, in der Regel mit dem Wert $-1$). Die Funktion `puts()` gibt einen String auf die Standardausgabe aus, im Gegensatz zu `fputs()` wird am Ende noch ein Zeilenumbruch ausgegeben.

### 7.3.3. Formatiertes Lesen und Schreiben

```
#include <stdio.h>

int printf(const char *format, ...);
int fprintf(FILE *stream, const char *format, ...);
int sprintf(char *str, const char *format, ...);
int snprintf(char *str, size_t size, const char *format, ...);
int scanf(const char *format, ...);
int fscanf(FILE *stream, const char *format, ...);
int sscanf(const char *str, const char *format, ...);
```

Die Funktionen `fprintf()` und `fscanf()` arbeiten wie `printf()` bzw. `scanf()` versehen mit der zusätzlichen Angabe, aus welcher Datei bzw. in welche Datei die Daten kommen – für `printf` und `scanf` sind `stdin` bzw. `stdout` eben festgelegt.

Die String-Varianten `sprintf()`, `snprintf()` und `sscanf()` schreiben in einen bzw. lesen aus einem String, was in Verbindung mit stringweiser Ein- und Ausgabe nützlich sein kann. Bei `snprintf()` wird zusätzlich die maximale Zahl an Zeichen (inklusive `'\0'`), die in `str` geschrieben werden kann, angegeben.

Näheres zu den Formaten findet sich in den Abschnitten C.2 und C.3.

### 7.3.4. Blockweises Lesen und Schreiben

```
#include <stdio.h>

size_t fread(void *ptr, size_t size, size_t nmemb,
             FILE *stream);
size_t fwrite(const void *ptr, size_t size, size_t nmemb,
              FILE *stream);
```

Ein `double` besteht aus `sizeof(double)` Bytes, typischerweise 8; gibt man ein `double` in voller Genauigkeit aus, in diesem Beispiel mit 16 gültigen Stellen plus Exponent, kommen mehr als doppelt so viele Bytes zum Einsatz, beschränkt man sich auf nur acht Stellen ohne Exponent, leidet die Genauigkeit. Die bitweise Darstellung der elementaren Datentypen ist in der Regel die

[1] Auch `fgets()` ist nicht ganz ohne Tücken: Sollte die Datei keine echte Textdatei sein (siehe unten), dann ist es möglich, dass `fgets()` auch Zeichen mit dem Wert 0 einliest (bis hin zum Ende der Zeile, der Datei oder des Arrays), d.h. man weiß nicht, ob das vermeintliche Stringende auch das tatsächliche ist, weil `fgets()` nicht die Anzahl der gelesenen Zeichen zurückgibt. Die beste Methode findet sich in Anhang C.3.1.

effizienteste. Wenn man seine Daten nur auf einem einzigen System verwendet, kann man alles in der Speicherdarstellung verwenden.

Das lässt sich mit Hilfe der Funktionen `fread()` und `fwrite()` realisieren, die in einen/aus einem bei `ptr` beginnenden Speicherbereich `nmemb` Objekte der Größe `size` aus dem/in den Dateistrom `stream` schreiben/lesen. In obigem Beispiel würde also

```
fwrite(&meindouble, sizeof(double), 1, stream);
```

das längere

```
fprintf(stream, "%16.15e ", meindouble);
```

ersetzen.

Die Textvariante ist allerdings insofern sicherer, dass auf anderen Systemen im Rahmen der Genauigkeit der Wert ebenfalls eingelesen werden kann, während man im binären Fall ein eigenes maschinenunabhängiges Format definieren müsste, in dem Werte gespeichert werden, wenn man ganz sicher gehen will. Außerdem lassen sich Textfiles leichter „von Hand“ editieren.

Die binäre Ein- und Ausgabe ist vor allem für komplexe Datentypen gedacht. Vorsicht ist hier in Verbindung mit Zeigern geboten – liest man auch die Zeiger auf dieser Basis ein, so hat man schnell ein undefiniertes Verhalten. Es ist sinnvoll, das Abspeichern und Einlesen von Strukturen in spezielle Funktionen auszulagern, die Zeiger beim Speichern geeignet durch den Wert ergänzen oder im Falle einer verketteten Liste die Zeiger (auf Vorgänger und Nachfolger) ganz auslassen und beim Laden durch dynamische Speicherallokation bzw. Neuverkettung der Liste behandeln.

## 7.4. Position in der Datei

```
#include <stdio.h>

int fseek(FILE *stream, long offset, int whence);
long ftell(FILE *stream);
void rewind(FILE *stream);
int fgetpos(FILE *stream, fpos_t *pos);
int fsetpos(FILE *stream, fpos_t *pos);
```

Jede Lese- oder Schreibaktion verändert die aktuelle Position in der Datei. Mit Hilfe von `fseek()` kann man die aktuelle Position um `offset` gegenüber `whence` verändern, wobei `whence` die Werte `SEEK_SET` (ab Dateianfang), `SEEK_CUR` (ab momentaner Position) und `SEEK_END` (ab Dateiende) annehmen kann. Ein Offset von `0L` bedeutet keine Veränderung gegenüber `whence`.

Die aktuelle Position kann man mittels `ftell()` herausfinden.

Alternativ zu `fseek(stream, 0L, SEEK_SET)` kann man auch `rewind(stream)` verwenden, mit dem zusätzlichen Nebeneffekt, dass alle Fehlerflags wieder auf Null gesetzt werden, man also wieder mit einer unberührten Datei anfängt.

Wenn die Dateilänge den `long`-Zahlenbereich übersteigt (sprich, man i.d.R. bei mindestens 2GB-Dateien angelangt ist), ist die Verwendung von `fgetpos()` und `fsetpos()` sinnvoll, die die

Position in dem implementationsspezifischen Typ `fpos_t` abspeichern. Diese Funktionen können natürlich auch standardmäßig statt `ftell()` und `fseek()` verwendet werden. Allerdings besteht keine Möglichkeit, `fpos_t` selbst zu setzen.

Nach einem Aufruf von `fsetpos()`, `fseek()` oder `rewind()` ist es möglich, die Schreib-/Lese-Richtung zu ändern (siehe auch `fflush()`).

```
#include <stdio.h>

int feof(FILE *stream);
```

Wenn eine Leseoperation die aktuelle Position über das Dateiende hinausschieben würde, wird diese auf das Dateiende gesetzt und es wird ein Flag gesetzt, das *end of file* signalisiert. Würde jetzt `fgetc()` aufgerufen, so wäre der Rückgabewert `EOF`. Die Funktion `feof()` prüft, ob das Flag gesetzt ist und liefert einen Wert ungleich Null zurück, wenn ja. Vorsicht: Mit `feof()` kann man also nur herausfinden, dass man das Dateiende erreicht hat, nachdem man es schon erreicht hat! Wenn man das Ergebnis der entsprechenden Leseoperation bereits verwendet hat, ist es zu spät…

## 7.5. Vermischtes

### 7.5.1. Ausgabe erzwingen

```
#include <stdio.h>

int fflush(FILE *stream);
```

Wenn ein Programm nicht läuft, verwendet man gerne Kontrollausgaben, um herauszufinden, wo es schiefgeht bzw. bis wohin es gerade noch gutgeht. Weil die Ausgabe aber gepuffert ist, bekommt man nicht unbedingt die letzte Kontrollausgabe, sondern nur die letzte Kontrollausgabe, die den Ausgabepuffer zum Überlaufen gebracht hat bzw. ihn gefüllt hat. Um den Ausgabepuffer für einen Ausgabestrom zu leeren, wird die Funktion `fflush()` verwendet. Unter UNIX: Wird `fflush()` mit dem Argument `NULL` verwendet, so werden alle offenen Ausgabeströme (einschließlich `stdout` und `stderr`) „geflusht“.

Die Wirkung einer Anwendung von `fflush()` auf einen Eingabestrom ist nicht definiert – vom Löschen des Eingabepuffers über gar nichts bis hin zum 10 000-Volt-Stromstoß aus der Tastatur ist alles möglich.

Bei einem mit `'+'` geöffneten *Updatestrom* bestimmt die letzte Dateioperation die „Richtung“ des Stroms. Will man die Richtung von Schreiben auf Lesen wechseln, muss man entweder `fflush()` oder eine der Positionierungsfunktionen aufrufen, um sicherzustellen, dass alle geschriebenen Daten auch tatsächlich in die Datei geschrieben worden sind.

### 7.5.2. Dateien löschen

```
#include <stdio.h>

int remove(const char *pathname);
```

Mit Hilfe von `remove()` kann man Dateien (oder Verzeichnisse) löschen. Falls der angegebene Name nur ein Link auf die Datei ist, wird nur der Link gelöscht. Ist die Datei geöffnet, ist das Ergebnis implementierungsabhängig.

### 7.5.3. Dateien umbenennen

```
#include <stdio.h>

int rename(const char *old, const char *new);
```

Mittels `rename()` kann man Dateien (oder Verzeichnisse) umbenennen. Existiert der neue Name bereits, so ist das Ergebnis implementierungsabhängig.

### 7.5.4. Temporäre Dateien

```
#include <stdio.h>

char *tmpnam(char *s);
FILE *tmpfile (void);
```

Manchmal benötigt man temporäre Dateien, um ein Ergebnis erst einmal zwischenzuspeichern – nur, wie nennt man die temporäre Datei, damit sie nicht so heißt wie eine bereits existierende Datei? Die Funktion `tmpnam()` liefert einen noch nicht vorhandenen Dateinamen, die Funktion `tmpfile()` öffnet eine temporäre Datei im Modus `"w+b"`, die nach dem Schließen automatisch gelöscht wird.

### 7.5.5. Status

```
#include <stdio.h>

void clearerr(FILE *stream);
int feof(FILE *stream);
int ferror(FILE *stream);
```

Mit Hilfe der Statusfunktionen kann man den aktuellen Status eines Stroms auslesen und ändern.

- `clearerr()` löscht alle Fehlerindikatoren sowie den end-of-file-Indikator.
- `feof()` liefert den Status des end-of-file-Indikators.
- `ferror()` liefert den Fehlerstatus und liefert Null zurück, wenn keine Fehler vorliegen.

Fehlercodes für die Datei bzw. alle Dateifunktionen sind in den manpages dokumentiert.

***Programmlisting 7.1:*** dateiops.c

```
/*****************************************************************
**
**  dateiops.c - Demo fuer High-Level-Datei-I/O
**
**
*****************************************************************/

/*** INCLUDES & DEFINES ******************************************/

#include <stdlib.h>  // exit()
#include <stdio.h>   // Datei-Ein-/Ausgabefunktionen

#define NUMBYTES  17
#define LINELEN   81


/*** EIGENE TYPEN ************************************************/

typedef struct {
    double d1;
    int d2;
    unsigned char d3[NUMBYTES];
} demodaten;


/*****************************************************************
**
**  main - Hauptprogramm
**
**  Oeffnet argv[1], kopiert es nach "ausgabe" und gibt es
**  mit Zeilennummern wieder aus. Oeffnet "ausgabe" anschliessend
**  zum Lesen und Schreiben, gibt den Inhalt aus, schreibt eine
**  Struktur binaer ans Ende und liest sie wieder ein.
**
*/

int main (int argc, char **argv)
{
    FILE *infile=NULL, *outfile=NULL;
    char buffer[LINELEN];
    int i;
    long offset;
    demodaten Daten[2] = {{1.0,2,{3,4}}, {-5.0,42,{0,1}}}, test;

    /* Datei argv[1] fuer Lesen oeffnen */
    if (argc==2)
        infile = fopen(argv[1], "r");

    /* Kein zweites Argument, falscher Dateiname,
    ** Fehler beim Oeffnen... */
    if (infile == NULL) {
```

```
        /* Formatierte Ausgabe nach stderr (weil Fehler) */
        fprintf(stderr, "Fehler beim Oeffnen von %s\n", argv[1]);
        fflush(stderr); // fflush() erzwingt die sofortige Ausgabe
        exit(EXIT_FAILURE);
    }

    /* Datei "ausgabe" fuer Schreiben oeffnen, bei Fehler
    ** wird Datei argv[1] geschlossen. Wenn "ausgabe" nicht
    ** existiert, wird es neu angelegt, ansonsten leer geoeffnet. */
    if ((outfile = fopen("ausgabe", "w")) == NULL) {
        fprintf(stderr, "Fehler beim Oeffnen von ausgabe\n");
        fflush(NULL); // NULL => "flush"t alle offenen Ausgabestroeme
        fclose(infile); // Datei mit dem Handle infile schliessen
        exit(EXIT_FAILURE);
    }

    for (i=1; fgets(buffer, LINELEN, infile) != NULL; i++) {
        fputs(buffer, outfile);
        fprintf(stdout, "%4d %s", i, buffer);
    }
    puts(""); // Aequivalent zu fputs("" "\n",stdout)

    fclose(infile); // Genug gelesen

    fclose(outfile); // Ab jetzt wollen wir schreiben _und_ lesen

    /* Mit r+ wird zusaetzlich zum Lesen das Schreiben ermoeglicht
    ** (Dateiinhalt vorhandener Dateien wird nicht geloescht),
    ** mit w+ wird zusaetzlich zum normalen Schreiben auch das
    ** Wiederauslesen des Geschriebenen ermoeglicht (Datei startet
    ** "leer", alte Inhalte sind verloren) */
    if ((outfile = fopen("ausgabe", "r+b")) == NULL) {
        fprintf(stderr, "Fehler beim Oeffnen von ausgabe\n");
        fflush(NULL);
        exit(EXIT_FAILURE);
    }

    fseek(outfile, 0L, SEEK_END); // Gehe zum Dateiende
    offset = ftell(outfile); // Aktuelle Position merken

    /* Datei zurueckspulen, i.e. zum Anfang gehen,
    ** Aequivalent zu: fseek(outfile, 0L, SEEK_SET)
    ** plus loeschen von Fehlerflags. */
    rewind(outfile);
    fprintf(stdout,"\nGebe nun die Datei \"ausgabe\" aus:\n");
    // Datei byteweise ausgeben:
    while ( (i=fgetc(outfile)) != EOF )
        fputc(i, stdout);

    // Wir sind jetzt wieder am Dateiende...

    // Binaere Ausgabe in die Datei
    fwrite(&Daten[0], sizeof(demodaten), (size_t) 2, outfile);

    // Wieder ans ehemalige Dateiende zurueckkehren...
    fseek(outfile, offset, SEEK_SET);
```

```
    // ... und den ersten Datensatz einlesen
    fread(&test, sizeof(demodaten), (size_t) 1, outfile);

    // Vergleichen
    if(Daten[0].d1!=test.d1||Daten[0].d2!=test.d2) {
        fprintf(stderr, "Habe nicht wiederbekommen, was ich ins"
                        " File geschrieben habe!\n");
    } else for (i=0; i<NUMBYTES; i++) {
            if(Daten[0].d3[i]!=test.d3[i]) {
                fprintf(stderr, "Habe nicht wiederbekommen, was ich"
                                " ins File geschrieben habe!\n");
                break;
            }
        }

    fclose(outfile);

    // Guter Stil
    exit(EXIT_SUCCESS);
}
```

# 8. Werkzeuge

An dieser Stelle ist der ***C***-Sprachumfang bis auf einige Details geklärt und die wichtigsten Funktionen der Standardbibliothek sind bekannt. Im Folgenden werden Werkzeuge betrachtet, die bei der Programmierung selbst Zeit sparen, strukturiertes Vorgehen ermöglichen oder bei der Fehlersuche hilfreich sind.

Dieser Überblick ist bei weitem nicht erschöpfend, weder in der Breite (Liste nützlicher Werkzeuge) noch in der Tiefe (genaue Beschreibung aller angeführten Programme).

## 8.1. gcc

Das wichtigste Werkzeug ist die Compiler-Umgebung selbst. Wer tiefer einsteigen will, sollte sich einen Überblick über den Funktionsumfang verschaffen. Einen Ausgangspunkt mag Abschnitt 3 geben. Einige Programme aus der *GNU Compiler Collection* werden im Folgenden noch separat betrachtet.

## 8.2. make

Das Programm *make* ist ein Werkzeug, das die Abarbeitung von Arbeitsschritten systematisiert. Es kann eingesetzt werden, wenn Skripte zu unübersichtlich werden bzw. wenn sehr viele gleichartige Regeln abgearbeitet werden müssen und oft nur Teile des Gesamtprojekts neu erstellt werden müssen.

Das ist beim Kompilieren und Linken großer Programmpakete der Fall.

*make* erwartet beim Aufruf eine Datei mit dem Namen `Makefile` bzw. `makefile` im aktuellen Verzeichnis. Hat das *Makefile* einen anderen Namen, muss dieser explizit übergeben werden.

Als weiteres Argument kann der Name eines Ziels (eines sogenannten *Targets*) des Aufrufs übergeben werden, z.B. `make clean`, `make all` u.ä.

Makefiles bestehen aus fünf Sorten von Objekten:

- *Explizite Regeln:* Wann und wie sollen eine oder mehrere Dateien (wieder-)erzeugt werden? Sie listen andere Dateien auf, von denen die Zieldatei(en) abhängen, die sogenannten *Voraussetzungen* und geben evtl. auch Befehle an, die zur Erzeugung oder Aktualisierung der Zieldatei(en) benötigt.

- *Implizite Regeln:* Besagen, wie eine Klasse von Dateien (festgelegt durch ein Namensmuster) aktualisiert/erstellt wird.

- *Variablendefinitionen:* Zeilenanweisung, in der einer Variable ein Textwert zugewiesen wird, der später eingesetzt werden kann.

- *Direktiven:* Spezielle Anweisungen an *make*, z.B. anderes Makefile einlesen oder Bedingungen auswerten, die dazu führen, dass bestimmte Teile des Makefiles ignoriert oder gelesen werden.

- *Kommentare:* Das Kommentarzeichen ist # und erzeugt einen Zeilenkommentar (wie `//` in ***C***).

Zeilen in Makefiles können durch einen Backslash am Zeilenende fortgesetzt werden.

Regeln sind folgendermaßen aufgebaut:
*Ziel(e)* : *Voraussetzungen*
*Befehl*
...
oder *Ziel(e)* : *Voraussetzungen*; *Befehl*
*Befehl*
...

Der freie Raum vor der zweiten und allen folgenden Zeilen *muss* vorhanden und durch einen Tabulatorsprung erzeugt sein. Leerzeichen reichen nicht!

Ein *Ziel* (oder *target*) ist im Normalfall ein Dateiname, kann aber auch ein „klingender“ Name, z.B. `clean` sein.

Die allererste Regel im Makefile ist das *Standardziel*. Dieses wird erstellt, wenn *make* ohne Zielangabe aufgerufen wird.

*make* geht folgendermaßen vor, um ein Ziel zu erstellen: Es überprüft die Voraussetzungen: Wenn eine der Voraussetzungen nicht mehr aktuell ist, wird sie neu erstellt, dann muss auch das Ziel neu erstellt werden. Andernfalls wird überprüft, ob die Zieldatei vorhanden ist und ob sie neuer als die Voraussetzungen ist. Ansonsten wird das Ziel aus den Voraussetzungen erstellt. Bei symbolischen Namen wie `clean` wird das Ziel demnach immer erstellt (es sei denn, es gäbe eine Datei namens clean).

Beispiel: Wir wollen das Präprozessor-Ergebnis sehen für die Dateien `main.c`, `m1.c`, `m2.c`, die ihrerseits von `meinedefs.h`, `m1.h` und `m2.h` abhängen:

```
    # Makefile

    help:
        echo "Mein tolles Makefile."
        echo "Targets: help (Standard)  prep  main.i  m1.i
m2.i"


    prep: main.i m1.i m2.i

    main.i: main.c meinedefs.h m1.h m2.h
```

```
    gcc -E main.c -o main.i

m1.i: m1.c meinedefs.h m1.h
    gcc -E m1.c -o m1.i

m2.i: m2.c m2.h
    gcc -E m2.c -o m2.i
```

Die Ziele `help` und `prep` erzeugen keine Dateien, sind also symbolisch zu verstehen. Die anderen Ziele führen dagegen zur Erzeugung der entsprechenden Dateien. Das Ziel `main.i` wird folgendermaßen behandelt: Das Vorhandensein der Dateien `main.c`, `meinedefs.h`, `m1.h` und `m2.h` wird überprüft; diese sind alle vorhanden. Wäre eine davon nicht vorhanden, so würde *make* nach einer Regel suchen, um sie zu erzeugen, und würde sie dann entweder erzeugen oder eine Fehlermeldung auswerfen. Da alles in Ordnung ist, werden nun die nachfolgenden Zeilen (in diesem Falle eine) ausgeführt.

Mit Hilfe von Variablen kann man das mehrmalige Tippen der Objekte vermeiden; außerdem kann durch ein vorangestelltes `@` die Ausgabe des gerade ausgeführten Befehls vermieden werden.

```
# Makefile

ifiles = main.i m1.i m2.i

help:
    @echo "Mein tolles Makefile."
    @echo "Targets: help (Standard)  prep  $(ifiles)"

prep: $(ifiles)

main.i m1.i: meinedefs.h m1.h
main.i: main.c m2.h
    gcc -E main.c -o main.i

m1.i: m1.c
    gcc -E m1.c -o m1.i

m2.i: m2.c m2.h
    gcc -E m2.c -o m2.i
```

Mittels `$(`*Variable*`)` erhält man den Wert einer Variablen. Voraussetzungszeilen können auch mehrmals vorkommen, wie man sieht. Das ist insbesondere nützlich, wenn in einer Zeile der Wert einer Variable geändert werden soll, und in der anderen dann tatsächlich Voraussetzungen stehen.

Prinzipiell kann man das Ganze auch kürzer schreiben:

```
# Makefile

ifiles = main.i m1.i m2.i
headers = meinedefs.h m1.h m2.h
```

```
units = main.c m1.c m2.c

help:
    @echo "Mein tolles Makefile."
    @echo "Targets: help (Standard)  prep  $(ifiles)"

prep: $(ifiles)

$(ifiles): $(units) $(headers)
    gcc -E main.c -o main.i
    gcc -E m1.c -o m1.i
    gcc -E m2.c -o m2.i
```

bzw.

```
....

prep: $(units) $(headers)
    gcc -E main.c -o main.i
    gcc -E m1.c -o m1.i
    gcc -E m2.c -o m2.i
```

Nachdem immer das gleiche passiert, kann man das auch allgemeiner schreiben:

```
# Makefile

CC = gcc
OPTIONS = -E
....

prep: $(units) $(headers)
    $(CC) $(OPTIONS) main.c -o main.i
    $(CC) $(OPTIONS) m1.c -o m1.i
    $(CC) $(OPTIONS) m2.c -o m2.i
```

Will man nun den Compiler oder die Flags verändern, ist das einfach möglich.

Die Verwendung von impliziten Regeln ermöglicht eine weitere Vereinfachung:

```
....

prep: $(ifiles) $(headers)

%.i: %.c
    $(CC) $(OPTIONS) $< -o $@
```

Die implizite Regel besagt: Baue eine Datei, die auf .i endet, wenn eine passende .c-Datei vorhanden ist, nach dem unten angegebenen Muster. Hier werden auch die automatischen Variablen $@ (Target) und $< (erste Voraussetzung) verwendet. Diese hätte man auch im allerersten Fall verwenden können:

```
....

main.i: main.c meinedefs.h m1.h m2.h
    gcc -E $< -o $@

m1.i: m1.c meinedefs.h m1.h
    gcc $(OPTIONS) $< -o $@

m2.i: m2.c m2.h
    $(CC) $(OPTIONS) $< -o $@
```

Deklariert man die symbolischen/klingenden Ziele als *phony targets*, kann es keine Probleme geben, wenn es eine Datei vom gleichen Namen gibt.

```
....

.PHONY: prep
prep: $(ifiles) $(headers)

....
```

Wenn ein Fehler auftritt, bricht *make* normalerweise ab. Das kann man durch ein vorangestelltes `-` verhindern:

```
....

clean:
    -rm -f *.i
```

## 8.3. lint/splint

Oft hat man das Problem, dass ein Programm trotz strenger Compilerwarnungen (im Fall von *gcc* bieten sich `-Wall -std=c99 -pedantic` an) problemlos kompiliert, aber trotzdem nicht funktioniert: Die Syntax ist hundertprozentig richtig, aber die Semantik ist falsch.

Mit *lint*[1] bzw. seinem aktuellen Nachfolger *splint* [2] kann man einen Schritt weitergehen: Es handelt sich um statische Semantikanalyse-Programme, d.h. Programme, die ohne das Programm laufen zu lassen, herauszufinden versuchen, ob das, was der Programmierer geschrieben hat, auch das ist, was er vermutlich gemeint hat.

Gewarnt wird unter anderem bei

- unbenutzten Variablen,
- nicht-expliziten Typkonvertierungen (Integer statt Wahrheitswert bei `if` u.ä.),

[1] engl. lint: Fussel, Flusen; diese sollen im Programm gefunden werden
[2] „Secure Programming Lint“, engl. to splint: schienen

- Zeigern, die bei Verwendung auch `NULL` sein könnten,
- möglicherweise überflüssigem Semikolon,
- möglicherweise unendlichen Schleifen,
- gefährlichen Makro-Konstruktionen,
- Speicherlecks (auf Wunsch – und nicht perfekt!).

Ein einfaches Beispiel für die Nützlichkeit von Splint liefert das folgende Programm[3]:

```
#include <stdio.h>

int main (int argc, char **argv)
{
    int a=0;

    if(a=4);
        return 0;
}
```

Wieviele potenzielle Fehler verbergen sich hier?

```
mairml@cip20:~/test> /opt/gcc340/bin/gcc -Wall -pedantic
 -std=c99 test2.c
test2.c: In function ‘main’:
test2.c:7: warning: suggest parentheses around assignment used
  as truth value
```

Macht Sinn – die Zuweisung ist ganzzahlig, erwartet wird ein Wahrheitswert. Außerdem schaut man nochmal drauf und stellt vielleicht fest, dass es ein `==` hätte werden sollen. *splint* findet dagegen ein wenig mehr:

```
mairml@cip20:~/test> splint test2.c
Splint 3.1.0 --- 22 Apr 2004

test2.c: (in function main)
test2.c:7:5: Test expression for if is assignment expression:
  a = 4
  The condition test is an assignment expression. Probably,
  you mean to use == instead of =. If an assignment is
  intended, add an extra parentheses nesting (e.g.,
  if ((a = b)) ...) to suppress this message. (Use -predassign
  to inhibit warning)
test2.c:7:5: Test expression for if not boolean, type int:
  a = 4
  Test expression type is not boolean or int. (Use
```

[3] Schamlos abgekupfert von *Linux-Magazin 05/2003, S. 90*

```
  -predboolint  to inhibit warning)
test2.c:7:10: Body of if statement is empty
  If statement has no body. (Use -ifempty to inhibit warning)
test2.c:3:15: Parameter argc not used
  A function parameter is not used in the body of the
  function. If the argument is needed for type compatibility
  or future plans, use /*@unused@*/ in the argument
  declaration. (Use -paramuse to inhibit warning)
test2.c:3:28: Parameter argv not used

Finished checking --- 5 code warnings
```

Es warnt vor dem möglicherweise aufgetretenen Vergessen von `==`, dem Missbrauch eines Ganzzahldatentyps als Wahrheitswert, der Tatsache, dass wir ein Semikolon direkt nach der `if`-Anweisung haben sowie dem Nicht-Benutzen der Argumente von `main`.

Neben den Warnungen wird auch jeweils eine Möglichkeit genannt, die entsprechenden Warnungen zu unterdrücken. Hierbei unterscheidet sich *splint* in der Parameterliste etwas von Standard-UNIX-Programmen:
`splint modus flags -f voreinstellungen dateiname`
Die Flags beginnen mit `+`, wenn etwas aktiviert wird, und mit `-`, wenn etwas deaktiviert wird. Der Modus legt fest, wie intensiv *splint* arbeiten soll, hierbei gibt es die Stufen Weak, Standard, Checks und Strict (für das erste *reale* Programm, das diese Stufe ohne Warnung schafft, ist eine Belohnung ausgesetzt…) Der Aufruf `splint -weak test2.c` produziert nur noch die erste und dritte Warnung, was für unsere Zwecke schon völlig ausreicht.

Eine weitere interessante Möglichkeit, mit *splint* besser zusammenzuarbeiten, ist im vierten Kommentar genannt: Man kann *splint*-spezifische Kommentare schreiben, die die Rolle von Variablen, Funktionen und Argumenten beschreiben, etwa `/*@unused@*/`, um mitzuteilen, dass man sich der Tatsache bewusst ist, dass eine Variable/ein Argument (noch) nicht benutzt wird. Das ist auch strukturell nützlich, weil man sich selbst noch einmal Gedanken darüber machen muss, wie eine Variable eingesetzt wird und was schiefgehen kann. Erstellt man die Kommentare während der Programmierung, ist der Aufwand vernachlässigbar. Eine Liste der möglichen Kommentare erhält man mit `splint -help annotations`. Für den Anfang sind neben `/*@unused@*/` meistens folgende Anmerkungen nützlich:

- In Funktionsdeklarationen kann man mit `/*@modifies@*/` kennzeichnen, welche Argumente durch die Funktion verändert werden (dürfen) – wer konsequent `const` verwendet, hat das indirekt eigentlich schon im Code stehen, genauso auch für
- die Anmerkung `/*@constant@*/`.
- *splint* überprüft die übergebenen Datentypen sehr rigoros, daher ist es sinnvoll, alle erlaubten Parameter-Typen anzugeben mittels der Alternativen-Anmerkung:
  `/*@alt zusaetzlicherTyp @*/`, z.B. `int /*@alt char@*/` akzeptiert auch `char`-Variablen statt eines echten `int` ohne Cast.
- Normalerweise wird gewarnt, wenn ein Zeiger auch den Wert `NULL` annehmen kann bzw. annimmt. Man kann aber den Null-Status auch für *splint* kennzeichnen:
  `/*@null@*/` – kann `NULL` werden; `/*@notnull@*/` – darf nie `NULL` werden.

- Ähnlich wie `/*@unused@*/` kann man auch extern definierte Variablen mittels `/*@external@*/` kennzeichnen.

- Manchmal stellt man ans Ende einer Funktion eine `return`-Anweisung, die alle Fehler abfangen soll, aber eigentlich nie erreicht wird. Um eine entsprechende Warnung von *splint* zu vermeiden, werden solche Anweisungen mit der Anmerkung `/*@notreached@*/` versehen.

- Ein Zeiger mit der Anmerkung `/*@only@*/` ist der einzige (bzw. primäre) Zeiger auf das entsprechende Objekt – in ***C***99 kann man das auch durch Verwendung von `restrict` erreichen. Zur Speicherfreigabe sollten ausschließlich `/*@only@*/`-Zeiger verwendet werden.

- Soll ein `/*@only@*/`-Zeiger doch kurzfristig kopiert werden, um etwa den allokierten Speicherbereich durchlaufen zu können, sollte der entsprechende Zeiger durch `/*@temp@*/` gekennzeichnet werden.

- Zeiger, die nie freigegeben werden, kann man durch `/*@shared@*/` kenntlich machen.

- Mithilfe von `/*@abstract@*/` kann man mitteilen, dass ein Datentyp, der per `typedef` erzeugt wurde, außerhalb einer Übersetzungseinheit/eines Moduls nie „ausgepackt“ wird, so dass man die Typdefinition problemlos ändern kann, ohne dass sich das für andere Module auswirkt (Beispiel: `FILE` – man weiß zwar, was alles drin steht, aber nicht genau, wo es steht und wie es in der Struktur heißt). Das Gegenstück dazu ist `/*@concrete@*/` – andere Module kennen mindestens einen Teil der Repräsentation des Typs, Änderungen am Typ betreffen auch alle anderen Module.

- Rückkehrverhalten von Funktionen: `/*@noreturn@*/`, `/*@maynotreturn@*/`, `/*@noreturnwhentrue@*/`, `/*@noreturnwhenfalse@*/`, `/*@alwaysreturns@*/`.

- Ob das Ergebnis wahr oder falsch ist, wenn ein Parameter `NULL` ist, kann mittels `/*@nullwhentrue@*/` und `/*@falsewhennull@*/` markiert werden.

Die ***C***99-Unterstützung von *splint* ist relativ gut, aber manche Dinge werden noch nicht unterstützt. Mithilfe des Flags `gnu-extensions` kann man auch eine Unterstützung für zumindest Teile der *gcc*-Erweiterungen sowie dazu kompatible Microsoft-Erweiterungen an- und ausschalten.

Für große Codes ist es sinnvoll, Namenskonventionen für Variablen-, Funktionen-, Makro- und Typnamen einzuführen. *splint* unterstützt auch das.

Bestimmte Tests können auch für bestimmte Code-Bereiche gezielt an- und ausgeschaltet werden, z.B.

```
....
/* Dieser Code sollte nie erreicht werden, aber man will ja
** sicher gehen... Test aus: *//*@ -unreachable @*/

fprintf(stderr,"\n\t**** %s, %ld (Funktion %s()):"
               " Murphy hat zugeschlagen!\n\n",
```

```
                __FILE__, (long) __LINE__, __func__);
  return (MYMODULE_UNREACH_ERRORCODE);
  /* Test wieder an: *//*@ +unreachable @*/
```

## 8.4. ar

Die Erzeugung eigener Bibliotheken wurde bereits in den Abschnitten 3.2 und 6.8 angesprochen. Auf UNIX-Systemen wird diese mit Hilfe des Binär-Archivierungsprogrammes *ar* durchgeführt. *ar* hängt im Wesentlichen nur die vom Compiler erzeugten Objektdateien aneinander und schreibt einen Index von definierten Funktionssymbolen in die Datei. Wenn man möchte, kann man die Objektdateien auch wieder extrahieren. *ar* ist in der Lage, auch einzelne Objektdateien zu ersetzen. Die *ar*-Kommandozeile sieht im Wesentlichen folgendermaßen aus:

```
ar Operation[Modifikatoren [RelPos] [Anzahl]] Archiv
    [Element...]
```

wobei mit „Elemente“ in diesem Fall die Objektdateien gemeint sind. Die wichtigsten Operationen sind

**d** Löschen,

**p** Ausgeben,

**r** Einfügen (mit Ersetzung),

**x** Extrahieren.

Diese können unter anderem modifiziert werden durch

**a, b, i** Position (an der eingefügt wird): hinter bzw. vor dem in `RelPos` angegebenen Element,

**N** Anzahl: falls es mehrere Instanzen eines Elements gibt, werden bis zu `Anzahl` von der Operation (löschen oder extrahieren) betroffen,

**s** Objektdateien-Index schreiben,

**u** Update: Normalerweise werden alle unter `Element...` angegebenen Dateien eingefügt, mit **u** nur die, die nicht vorhanden oder aber neuer als eine bereits vorhandene Datei sind.

**v** Mehr Ausgabe.

In einem Makefile könnte der Teil für die Archivierung in eine Bibliothek etwa folgendermaßen aussehen:

```
AR=ar
ARFLAGS=rus
LIB_OBJ=modul1.o modul2.o modul3.o modul4.o modul5.o \
    modul6.o
```

```
LIB_NAME = libmeinebib.a
....

$(LIB_NAME): $(LIB_OBJ)
    $(AR) $(ARFLAGS) $(LIB_NAME) $(LIB_OBJ)
```

Wer eigene Bibliotheken baut, sollte übrigens nicht vergessen, die notwendigen Header-Files ebenfalls zugänglich zu machen. In der Regel geschieht dies durch Sammeln der Header-Dateien in einem Verzeichnis und Aufnahme dieses Verzeichnisses in den Include-Pfad mittels der Compiler-Option `-I` *Include-Pfad*. Die Bibliothek wird entweder bei den anderen Bibliotheken abgelegt oder ebenfalls durch explizite Pfadangabe mittels `-L` *Bibliotheks-Pfad* zugänglich gemacht.

## 8.5. cvs

Wer mit mehreren anderen Programmierern zusammenarbeitet, stößt schnell auf das Problem, dass verschiedene Leute an der gleichen Datei arbeiten wollen oder vergessen, geänderte Dateien allen zugänglich zu machen, oder dass Änderungen zu einander inkompatibel sind u.ä.

Abhilfe schaffen sogenannte Versions-Management-Systeme, die *alle* Versionen, die jemals ins Repositorium eingestellt wurden, aufheben und melden, wenn lokale Versionen der einzelnen Programmierer dazu unterschiedlich oder inkompatibel sind. Mitunter ist es auch möglich, eine Datei für alle anderen zu sperren, so dass sichergestellt ist, dass nur man selbst Änderungen einspielen kann.

Ein frei verfügbares Versions-Management-System ist *cvs* (*concurrent version system*). Hier soll nicht näher auf die Syntax eingegangen werden.

## 8.6. gdb und ddd

Segmentation fault. Spätestens, wenn das Programm mit einem unerlaubten Speicherzugriff endet, gilt es, sich an die Fehlersuche zu machen. Bis jetzt kennen wir hierfür mehrere Möglichkeiten:

- Compiler-Warnungen beachten. Manchmal vergisst man das, aber viele Fehler verschwinden, wenn man die Compiler-Warnungen (`-Wall -pedantic`) abgearbeitet hat.
- `printf`-Zugang: Den Code mit `fprintf(stderr, ....); fflush(stderr);` pflastern, zunächst nur, um die Stelle einzukreisen, wo der Fehler auftritt, dann, um die Werte von Variablen auszugeben, die am Fehler beteiligt sind.
- *splint* benutzen. Mächtiger als Compiler-Warnungen und bei „Standardfehlern“ oft schneller als die `printf`-Methode.
- Jemanden fragen, der sich besser auskennt (legitimes Mittel).

Die erste und dritte Methode sind statisch, also ohne Kenntnis der im Programm angenommenen Variablen-Werte, die letzte mit dem Makel der mitunter mangelnden Verfügbarkeit behaftet.

Mit der `printf`-Methode kann man jeden Programmierfehler aufspüren – mit einem entscheidenden Nachteil: Es ist mitunter sehr, sehr mühsam.

Mit Hilfe eines Debuggers kann ein Gutteil der Nachteile des `printf`-Zugangs abgemildert werden. Der GNU-Debugger *gdb* kann die folgenden Aufgaben erledigen:

- Das Programm starten und gegebenenfalls Parameter übergeben; das Programm läuft bis zum Abbruch/Ende bzw. bis zum ersten Breakpoint: `run [Kommandozeile]`
- Setzen von *Breakpoints*, an denen die Programmausführung unterbrochen wird. Weiß man etwa, dass der Fehler erst in der Funktion `xyz()` aufgetreten ist, muss man nicht Schritt für Schritt durchs Programm gehen, bis man dort angekommen ist, sondern setzt einen Breakpoint auf den Anfang der Funktion (etwa mittels `break xyz` oder `break dateimitxyz.c:xyz`) und sagt dem Debugger, er soll die Programm-Ausführung ganz normal laufen lassen bis er an einen Breakpoint kommt oder das Programm zu Ende ist.
  Breakpoints können mittels `cond Breakpoint-Nummer Bedingung` mit einer Bedingung versehen werden (etwa in einer Schleife `i>=1000`).
- Fortfahren bis zum nächsten Breakpoint oder bis zum Ende: `continue`
- Eine Zeile weitergehen innerhalb der aktuellen Funktion (nicht Absteigen in aufgerufene Funktionen): `next`
- Eine Zeile weitergehen innerhalb der aktuellen Funktion oder gegebenenfalls Absteigen in die aufgerufene Funktion: `step`
- Den Wert einer Variable ausgeben: `print`
- Die Debug-Session abbrechen: `quit`
- Hilfe zu den Möglichkeiten erfragen: `help`
- Den Aufruf-Stack ansehen: `bt` (Backtrace)

Ein Programm muss mit der Option `-g` (oder einer ihrer mächtigeren Verwandten) kompiliert werden, damit es Debug-Symbole und den Hinweis auf die Quelldatei in der Programmdatei gibt.

Der *gdb*-Aufruf ist: `gdb programm`

Eine typische Fehlersuche ausgehend von einem segfault könnte folgendermaßen aussehen:

1. Eigenen Debug- oder Testcode mittels der `-D`-Option des Compilers aktivieren. Wenn das nichts hilft,
2. *gdb* anwerfen, Programm starten. An der Stelle, an der der segfault auftritt, mittels `where` herausfinden, an welcher Stelle im Programm man sich befindet, von welcher Funktion die aktuelle Funktion an welcher Stelle aufgerufen wurde, von welcher Funktion die aufrufende Funktion an welcher Stelle aufgerufen wurde usw. bis hoch zu der Zeile in `main`, von wo aus die vorletzte Funktion in der Reihe aufgerufen wurde. Jetzt weiß man schon relativ genau, ab wo es schiefgehen könnte. Nun kann man einen Breakpoint genau auf die Zeile setzen, deren Aufruf den segfault verursacht hat und das Programm wieder durchlaufen

lassen. Am Breakpoint gibt man sich alle Variablen-Werte aus und weiß dann (hoffentlich), welche Variable schuld ist. Gegebenenfalls muss man das Ganze mit `continue` fortsetzen/wiederholen, bis man an dem Aufruf dieser Stelle angekommen ist, der zum segfault führt.

3. Von hier aus kann man sich zu der Stelle vorarbeiten, die die entsprechende Variable falsch gesetzt hat, und den Fehler beheben.

Des weiteren kann man Variablen mittels `set` setzen bzw. auch Debugger-interne Variablen einführen, mittels `jump` in eine Zeile oder an eine Speicheradresse springen, wo die Programmausführung fortgesetzt werden soll, mittels `watch` an dem Punkt anhalten, wo sich eine Variable oder Speicherstelle geändert hat und mittels `attach` sogar einen bereits laufenden Prozess debuggen (der natürlich mit Debug-Informationen kompiliert worden sein muss).

Falls man einmal darauf angewiesen ist, in die Standard-Bibliothek hineinzugehen, um herausfinden zu können, wo etwas schiefgeht, so muss man sich eine Version der Standard-Bibliothek erstellen oder besorgen, die mit Debug-Symbolen versehen ist.

Mittlerweile gibt es einen grafischen Aufsatz für *gdb*, den *Data Display Debugger ddd*. Dieser zeigt im Gegensatz zum *gdb* in einem separaten Fenster den Quellcode der aktuellen Objektdatei am momentanen Ausführungspunkt (*gdb* zeigt nur jeweils die nächste auszuführende Zeile bzw. mit `list` ein bisschen was davor und danach) und bietet die Möglichkeit, Daten grafisch darzustellen: Durch den Befehl `graph display` *variable* werden die Variable und ihr Wert im Grafikfenster dargestellt. Im Falle von Zeigern expandiert ein Doppelklick auf den Wert, im Falle von Strukturen und Vereinigungen wird der Wert eines Eintrags angezeigt. Die wichtigsten Debugger-Befehle sind über ein schwebendes Menü bzw. aktuell interessante Befehle über das Kontext-Menü zu erreichen. Der Aufruf erfolgt wie bei *gdb*, tatsächlich läuft im unteren Teilfenster der *gdb*.

## 8.7. electric fence

Ein häufiger Grund für unerklärliches Programmverhalten ist das Hinausschreiben über die Grenzen von Feldern oder dynamisch allozierten Speicherbereichen hinweg, wodurch man oft in andere eigene Datenobjekte hineinschreibt.

Hierfür gibt es eine Menge spezialisierter Debug-Tools, z.B. *purify* oder oder das mächtige *valgrind* und auch spezielle Compiler-Versionen wie *checkergcc*. Eines der ältesten und am einfachsten anzuwendenden Werkzeuge ist *electric fence* (kurz *efence*). Zudem ist es – im Gegensatz zu *purify* – freie Software.

Die Arbeitsweise von *efence* ist relativ übersichtlich: Es ersetzt die Speicherallozierung des Betriebssystems durch eine eigene Variante, die nach dem allozierten Speicher eine nichtzugängliche Speicherseite platziert. Wenn nun in diese Speicherseite hineingeschrieben wird, dann hält *efence* sofort die Programmausführung mit einem segfault an und man weiß, wo genau es schiefgeht. Ein offensichtlicher Nachteil ist der potenziell hohe Speicherverbrauch.

Die Anwendung ist recht einfach: Programm bauen, mit der Bibliothek `libefence.a` linken, im Debugger starten und auf den segfault warten. Für den Fall, dass keiner kommt, kann man in der Shell oder mit dem Debugger die Variable `EF_PROTECT_BELOW` auf 1 setzen, um statt

einer Speicherseite nach dem allozierten Bereich eine Speicherseite davor zu überwachen. Es ist auch durch das Setzen von `EF_PROTECT_FREE` möglich, mittels `free()` freigegebenen Speicher weiter zu überwachen (für den Fall, dass das Programm auf bereits freigegebenen Speicher zugreifen will).

Im Abschnitt über Datentypen und Speicher, 4.18, ist bereits erklärt worden, dass bestimmte Variablentypen an bestimmten Wortgrenzen (Vielfachen von Speicherwortgrößen, z.B. von 2, 4 oder 8) ausgerichtet werden müssen. In ähnlicher Weise müssen allozierte Speicherbereiche und auch die geschützte Seite an einer solchen Grenze beginnen. Das kann zur Folge haben, dass zwar eine Verletzung von Speichergrenzen vorliegt, *efence* sie aber nicht detektieren kann, weil sie z.B. nur die drei Bytes vor der Wortgrenze zum geschützten Speicherbereich betrifft. In der Regel bedeutet dies, dass der Fehler (zumindest auf diesem System) zunächst auch keinen Schaden anrichtet – bei einer Änderung der allozierten Speichergröße ist das Ganze dann aber wieder von großem Interesse. Für diesen Fall gibt es auch noch die Möglichkeit, die Wortgrenzen aus *efence*-Sicht aufzuheben zu versuchen mittels des Setzens von `EF_ALIGNMENT` auf einen kleineren Wert (Standard: `sizeof(int)`), sinnvollerweise auf 1 bzw. 0, zu setzen. Auf manchen Systemen kann es auch zu „unechten“ segfaults kommen, wenn Funktionen (z.B. `strcmp()`) zu viele Bytes einlesen und dann über das Ende hinausschießen.

## 8.8. prof/gprof

Wenn ein Programm eine zeitkritische Aufgabe übernimmt oder ein Algorithmus möglichst effizient implementiert werden soll oder wenn ganz einfach der Verdacht besteht, dass ein Programm viel Zeit verschwendet, dann ist es sinnvoll, sich mit Hilfe eines *Profilers* wie *prof* oder *gprof* anzusehen, welche Aufruf-Abläufe stattfinden und wie oft einzelne Funktionen durchlaufen werden bzw. wie der Anteil dieser Funktionen an der Gesamtlaufzeit ist. Manchmal kann es auch sinnvoll sein herauszufinden, welche Zeilen am öftesten ausgeführt werden.

Die Programme *prof* und *gprof* sind Profiler, die man typischerweise auf UNIX-Betriebssystemen vorfindet. Im Folgenden wird nur auf die mit *gcc* mitgelieferte Variante von *gprof* Bezug genommen.

Nachdem in einem fertigen Programm in der Regel keine Symbole mehr stehen, die verdeutlichen, wo Funktionen oder gar einzelne Anweisungszeilen des Quellcodes waren, müssen die Programme speziell kompiliert und gelinkt werden. Bei *gcc* gibt es hierzu die Option `-pg`. In der Regel erstellt man kein Programm mit einem speziellen Namen, sondern belässt es bei `a.out`. Wird `a.out` aufgerufen, dann werden Laufzeitmessungen gestartet. Bei ordnungsgemäßer Programmbeendung durch `return` in `main` bzw. durch `exit` wird eine Datei `gmon.out` erstellt, die das Ergebnis der Laufzeitmessungen enthält. Der Aufruf von *gprof* bereitet dann die Ergebnisse auf. Wichtig: Will man nicht nur funktions- sondern auch zeilenweise Informationen, so muss das Programm auch mit der Option `-g` übersetzt werden.

**Beispiel:** Das Programm `heron.c` (siehe Ende des Kapitels) vergleicht eine Implementierung der HERONschen Methode[4] zur Berechnung einer Quadratwurzel mit einer rekursiven Implemen-

[4] Benannt nach HERON von Alexandria (vermutlich 2. Jhd.); er gibt Methoden zum quadratischen und kubischen Wurzelziehen an und berechnet die dritte Wurzel aus 100 auf einige Stellen. Die Methode zur Berechnung der Quadratwurzel war bereits auf den Gesetztafeln des HAMMURABI vermerkt und vermutlich schon zweitausend

tierung sowie auf Wunsch mit der Bibliotheksfunktion `sqrt()`. Wir wollen nun die Aufruf-Statistik der Funktionen wissen:

```
$ gcc -pg -g -DOUTPUT -lm heron.c
$ ./a.exe 7
sqrt(7) = 2.64575131106459 <-> 2.64575131106459 <-> 2.64575131106459
$ gprof -p -b
Flat profile:

Each sample counts as 0.01 seconds.
 no time accumulated

  %   cumulative   self              self     total
 time   seconds   seconds    calls  Ts/call  Ts/call  name
  0.00      0.00     0.00        1     0.00     0.00  mysqrt
  0.00      0.00     0.00        1     0.00     0.00  mysqrt_rec
  0.00      0.00     0.00        1     0.00     0.00  mysqrt_rec_iter
```

Die Funktionen laufen nicht langsam genug, um entsprechend Zeit anzusammeln, deshalb lassen wir in einer Schleife den Hauptteil von `main` oft genug durchlaufen; es ist übrigens sinnvoll für Profiling, einen nicht mit anderen Aufgaben betrauten Rechner zu verwenden, um die Ergebnisse nicht zu verfälschen.

```
$ gcc -pg -g -DPROFILING -DWIE_OFT=100000000 -DOUTPUT -lm heron.c
$ ./a.exe 7
sqrt(7) = 2.64575131106459 <-> 2.64575131106459 <-> 2.64575131106459
$ gprof -p -b a.exe
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ns/call  ns/call  name
 52.32     40.43    40.43 100000000   404.30   404.30  mysqrt_rec_iter
 38.63     70.28    29.85 100000000   298.50   298.50  mysqrt
  4.59     73.83     3.55                             main
  4.24     77.11     3.28 100000000    32.80   437.10  mysqrt_rec
  0.13     77.21     0.10                             sqrt
  0.08     77.27     0.06                             atoi
```

In der ersten Spalte ist der prozentuale Anteil an der Gesamtprogrammlaufzeit festgehalten. Die rekursive Variante kommt erheblich schlechter weg (56.56% im Vergleich zu 38.63%), die Bibliotheksvariante ist wie erwartet wesentlich schneller. Die Anzahl an Aufrufen von `mysqrt_rec_iter()` ist allerdings mit Sicherheit falsch. Die zeilenweise Auswertung gibt hier etwas mehr Aufschluss:

```
$ gprof -l -p -b a.exe
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ns/call  ns/call  name
 34.57     26.71    26.71                             mysqrt_rec_iter (heron.c:37 @ 4011b8)
 30.89     50.58    23.86                             mysqrt (heron.c:28 @ 401111)
 14.48     61.77    11.19                             mysqrt_rec_iter (heron.c:35 @ 4011a4)
  5.41     65.94     4.18                             mysqrt (heron.c:27 @ 4010fd)
  3.02     68.28     2.33                             main (heron.c:68 @ 401319)
```

Jahre vor HERON bei den Babyloniern bekannt, weswegen sie oft auch babylonisches Wurzelziehen genannt wird.

```
 2.15     69.94     1.66 500000000     3.32     3.32  _mysqrt_rec_iter (heron.c:34 @ 401188)
 1.79     71.32     1.39                          mysqrt_rec (heron.c:45 @ 40124e)
 1.75     72.67     1.35                          mysqrt (heron.c:22 @ 4010d6)
 1.18     73.58     0.91                          main (heron.c:69 @ 40134b)
 1.13     74.45     0.87                          mysqrt_rec_iter (heron.c:39 @ 40122c)
 0.96     75.19     0.74                          mysqrt_rec (heron.c:50 @ 401275)
 0.71     75.73     0.55 100000000     5.45     5.45  _mysqrt_rec (heron.c:42 @ 401238)
 0.65     76.23     0.50                          mysqrt_rec (heron.c:51 @ 40128a)
 0.40     76.55     0.31 100000000     3.10     3.10  _mysqrt (heron.c:19 @ 4010c0)
 0.21     76.70     0.16                          main (heron.c:65 @ 40130c)
 0.14     76.81     0.11                          mysqrt_rec (heron.c:48 @ 401263)
 0.13     76.92     0.10                          _sqrt (heron.c:88 @ 402060)
 0.11     77.00     0.09                          main (heron.c:70 @ 401359)
 0.08     77.06     0.07                          main (heron.c:72 @ 401367)
 0.08     77.13     0.07                          mysqrt (heron.c:25 @ 4010eb)
 0.08     77.19     0.06                          _atoi (heron.c:88 @ 402070)
 0.06     77.23     0.04                          mysqrt (heron.c:30 @ 401163)
 0.03     77.25     0.02                          mysqrt (heron.c:31 @ 401169)
 0.02     77.27     0.01                          mysqrt (heron.c:23 @ 4010e4)
```

Fünf Durchläufe des Verfahrens klingt realistischer. Der Grund für das falsche Zählen liegt darin, dass *gprof* mit statistischen Methoden arbeitet und deshalb manche Aufrufe verpasst. Hier erkennt man jetzt auch, dass der Hauptgrund für die Langsamkeit des Algorithmus die übermäßig aufwendig zu berechnende Abbruchbedingung ist. Wäre unser Startwert auf eine Stelle genau, so ließe sich wegen der quadratischen Konvergenz des Verfahrens die Schleife durch fünf Aufrufe des Updates ersetzen. Ein einfacher Test zeigt den Unterschied: Ersetzen wir das Update durch sechs Updates, so reduziert sich die Programmlaufzeit entsprechend (auf $47.47s$). Steckt man mehr Aufwand in die Berechnung eines guten Startwertes, so dass man sich die Abfrage schenken kann, so dürfte das Ganze noch schneller werden[5].

Bis jetzt haben wir nur eine der Möglichkeiten von *gprof* genutzt, es gibt aber noch weitere:

**-p** Gibt ein sogenanntes flaches Profil aus, Aufrufeinheiten (Funktionen bzw. Zeilen), Zeiten, Aufrufanzahl.

**-q** Ausgabe des Aufrufgraphen, der zeigt, welche Funktion von welcher anderen wie oft aufgerufen wurde.

**-A** Quellcode-Anmerkungen: Aufrufanzahl neben Funktionennamen.

**-y** Ausgabe der Anmerkungen in eine Datei (Quellcode-Dateiname plus `-ann` am Ende).

**-b** Ausgabe von Profil und Aufrufgraphen ohne Erklärungen.

**-l** Zeilenweises statt funktionsweises Arbeiten.

**-C** Zähle, wie oft welche Aufrufeinheit aufgerufen wurde. Wäre in Kombination mit -l ideal, funktioniert aber irgendwie nicht.

[5] Man kann etwa die Werte in $[1,4) * 2^{2k}$ in Werte in $[1,4)$ überführen mittels der Nicht-Standard-C-Funktion `ilogb()` und der Funktion `ldexp()`, dort einen guten Startwert ermitteln, oft genug iterieren und dann mittels `ldexp()` mit $2^k$ multiplizieren

## 8.9. Editor

Der Editor kann nicht nur zum Bearbeiten des Quelltextes dienen – auch wenn Syntax-Highlighting, Klammernprüfung, Einblenden von Zeilennummern, Falten von Code-Abschnitten usw. sehr nützlich sind.

Mächtige Editoren mit eigener Makro-Sprache und der Möglichkeit, Shell-Befehle aufzurufen, können beispielsweise den Compiler-Aufruf gleich mit übernehmen und bei Fehlern und Warnungen zur entsprechenden Zeile in der entsprechenden Datei springen.

Eine andere nützliche Sache sind sogenannte *ctags*. Hierbei erzeugt man mit einem speziellen Programm eine Übersicht der Funktions- und Makrodefinitionen und kann sich dann später auf Tastendruck die entsprechende Funktion ansehen. Das ist vor allem bei großen Programmierprojekten nützlich, wenn man nicht mehr unbedingt weiß, welche Funktion jetzt aus welchem Modul stammt.

## 8.10. Übersicht

| Name | Zweck | Quelle |
|---|---|---|
| gcc | Compiler(-Umgebung) | `gcc.gnu.org/` |
| gdb | (Zeilen-)Debugger | `www.gnu.org/software/gdb/` |
| ddd | Debugger-Frontend | `www.gnu.org/software/ddd/` |
| make | Build-Automatisierung | `www.gnu.org/software/make/` |
| ar | Archivierer | `www.gnu.org/software/binutils/` |
| gprof | Profiler | |
| nm | Symbole anzeigen | |
| strip | Symbole entfernen | |
| efence | `malloc`-Debugging | `perens.com/FreeSoftware/ (tot?)` |
| | | `packetstormsecurity.nl/UNIX/misc/index3.html` |
| splint | Code-Analyse | `www.splint.org` |
| exuberant ctags | ctags | `ctags.sourceforge.net` |
| Emacs | Editor | `www.gnu.org/software/emacs/` |
| NEdit | Editor | `www.nedit.org` |
| vim | Editor | `www.vim.org` |

**Tabelle 8.1.:** Werkzeuge und Bezugsquellen

In Tabelle 8.1 findet sich eine (bei weitem nicht vollständige) Übersicht nützlicher Software und von Bezugsquellen für Quellcode/ausführbare Dateien/Dokumentation.

***Programmlisting 8.1:*** heron.c

```
/*****************************************************************
**
**  heron.c -- Heronsche Regel zur Wurzelberechnung
**
*****************************************************************/

#include <stdlib.h>
#ifdef OUTPUT
#include <stdio.h>
#include <math.h>
#endif

#ifndef PRECVAL
#define PRECVAL 1e-15
#endif

#define ABS(I) ((I)>0 ? (I) : -(I))

double mysqrt (double base)
{
    double approx;

    if(base<=0.0)
        return 0.0;

    approx = (1.0+base)/2.0; // Wurzel liegt zw 1 und base/base und 1
    do {
        approx = 0.5*(approx+base/approx);
    } while (ABS((approx*approx/base-1.0))>PRECVAL);

    return approx;
}

double mysqrt_rec_iter (double base, double approx)
{
    approx = 0.5*(approx+base/approx);

    return (ABS((approx*approx/base-1.0))>PRECVAL) ?
            mysqrt_rec_iter(base,approx) : approx;
}

double mysqrt_rec (double base)
{
    double approx;

    if(base<=0.0)
        return 0.0;

    approx = (1.0+base)/2.0;

    return mysqrt_rec_iter(base,approx);
```

```
}

int main(int argc, char **argv)
{
    double radicand, erg, erg_rec;
#ifdef OUTPUT
    double erg_sqrt;
#endif
#ifdef PROFILING
#ifndef WIE_OFT
#define WIE_OFT 100000000
#endif
    int wie_oft = WIE_OFT;

    while (wie_oft--) {
#endif

    radicand = (argc==2) ? atoi(argv[1]) : 2.0;
    erg      = mysqrt(radicand);
    erg_rec  = mysqrt_rec(radicand);
#ifdef OUTPUT
    erg_sqrt = sqrt(radicand);
#ifdef PROFILING
    }
#endif
    printf("sqrt(%g) = %16.15g <-> %16.15g <-> %16.15g\n",radicand,
           erg, erg_rec, erg_sqrt);
#else
#ifdef PROFILING
    }
#endif

    if (erg != erg_rec)
        return 1;
#endif

    return 0;
}
```

# 9. Optimieren

## 9.1. Grundlagen

Wenn man im Hinblick auf Programmierung von Optimierung spricht, denkt man meistens an kürzere Programmlaufzeiten für komplexe und langwierige Programme. Das ist mit Sicherheit auch der Bereich, in dem die meisten Optimierungen stattfinden. Nichtsdestotrotz kann man im Umfeld der Programmierung leicht vier grundsätzliche Arten von Optimierung finden:

- Reduktion der Programmierzeit: Hierbei geht es nicht nur um kurzfristige Zeitgewinne, die häufig nur durch einen schlampigen Programmierstil geprägt sind, sondern auch um mittel- und langfristige Zeitgewinne durch einen entsprechenden Programmierstil, der auf Fehlervermeidung, Fehlererkennung, wiederverwendbaren Code u.ä. ausgelegt ist.
- Programmlaufzeit (ohne Interaktion mit dem Benutzer): Die reine Laufzeit des Programmes bzw. eines Teiles davon. Das ist der Teil, der sich messen lässt.
- Programmlaufzeit (Bedienzeit): Durch unübersichtliche, schlecht durchdachte, missverständliche oder unflexible Benutzerschnittstellen, sei es auf der Konsole oder auf einer grafischen Oberfläche, wird bei Anwendungen mit kurzer „echter“ Programmlaufzeit oft mehr Zeit verschwendet als durch das eigentliche Programm.
- Speicheraufwand: In vielen mathematischen oder ingenieurwissenschaftlichen Anwendungen ist der begrenzende Faktor der Speicher, weil eine Verdopplung der Problemauflösung eine Verdopplung in jeder Problemdimension bedeutet.

Bis auf die Benutzerschnittstellen wird im Folgenden auf einige Optimierungsansätze eingegangen. Während sich der Programmieraufwand nicht direkt bestimmen lässt, ist das für Zeit- und Speicheraufwand von Algorithmen leicht möglich. Noch schwerer ist der Zeitaufwand für die Bedienung eines Programmes zu fassen.

Wer Anwendungen für andere entwickelt, sollte deshalb frühzeitig herausfinden, was ihnen wichtig ist, ob es Standardeinstellungen oder Standardprofile gibt, auf die sie gerne schnell zugreifen können würden, mit welcher Art von Menüführung und -anordnung sie am besten klarkommen etc.

Darüber hinaus ist es selbst im nichtgewerblichen Bereich sinnvoll, eine Art Pflichtenheft zusammenzustellen, das die Anforderungen an die fertige Software genau beschreibt – insbesondere im Hinblick auf Grenzfälle, die das Programm noch behandeln können muss oder nicht mehr behandeln können muss. Eine langfristige Anwendungsperspektive ist ebenfalls sinnvoll, damit man bei der Programmentwicklung an den entsprechenden Stellen von vornherein flexible Lösungen vorsehen kann.

Ein vielversprechender Zugang, um einen guten Ausgangspunkt für Laufzeitoptimierung zu haben, soll hier vorweg erwähnt werden: Die drei „goldenen“ Regeln

1. Code so sauber wie möglich schreiben; auf Funktion überprüfen. Erst, wenn *alles* funktioniert, weitergehen zum nächsten Schritt.

2. Profiling für realistische Testfälle, um „hot spots“ zu finden, wo viel Zeit verloren geht.

3. Hot spots der Reihe nach bearbeiten. Jedes Mal überprüfen, ob die „Optimierung“ etwas gebracht hat und ob immer noch alles funktioniert. Gegebenenfalls Optimierung rückgängig machen/Code reparieren.

Ein guter Programmierstil spielt also als Ausgangsbasis eine wichtige Rolle, um überhaupt vernünftig optimieren zu können.

Was dieser Abschnitt nicht leistet, ist eine grundlegende Einführung in Software-Entwicklung oder Algorithmen-Theorie. Er soll nur eine Idee davon vermitteln, was möglich und sinnvoll ist bzw. an welchen Stellen man aufpassen sollte.

## 9.2. Programmieren optimieren

### 9.2.1. Erst denken, dann programmieren

Sobald man sich klar ist, was man programmieren soll, stellt sich die Frage, wie man es programmiert. Um das zu planen, ist es sinnvoll, sich eine Gesamtübersicht zu machen, die im einfachsten Fall aus Zeichnungen und Übersichten auf Schmierpapier besteht.

Bereits hier wird man Abhängigkeiten identifizieren können, die frühzeitig bekannt sein sollten, damit man nicht Teile des Codes wieder wegwerfen muss. Einige Grundlagen zum Thema Modularisierung wurden bereits im Abschnitt 3.3 besprochen, hier soll nun mehr auf die Methodik beim Aufteilen eingegangen werden.

Je nach Geschmack, Anwendung und Anzahl der Programmierer bieten sich unterschiedliche Vorgehensweisen bei der Implementierung an. Die beiden einfachsten sind die *bottom up*- bzw. *top down*-Programmierung. Im zweiten Fall geht man gewissermaßen von `main` aus und geht vom Abstrakten zum Elementaren: Man schreibt die wesentlichen Funktionen, in denen die verwendeten Algorithmen implementiert sind, als erstes, ohne sich darum zu kümmern, woher Daten, Optionen und Einstellungen kommen, evtl. sogar ohne sich um die Implementierung der Datentypen zu kümmern. Dann erst kümmert man sich, vom Abstrakten zum Elementaren um die tatsächliche Implementierung der Module, die nur Daten liefern, speichern oder elementar verarbeiten. Im Gegensatz dazu wird bei der bottom up-Programmierung überlegt, welche elementaren Aufgaben anfallen werden. Dann werden Module programmiert, die das und vielleicht noch mehr leisten. Anschließend gibt es unter Umständen eine abstrahierende Zwischenlage, die eine Schnittstelle zur tatsächlichen Anwendung bereitstellt und die elementaren Module unter Umständen schon sinnvoll verknüpft. Erst zum Schluss werden dann unter Benutzung der bereits entstandenen Bausteine die Algorithmen und das Hauptprogramm zusammengebaut.

Beiden Zugängen ist gemeinsam, dass das Programm oder Teilprogramm strukturiert wird und dann die strukturellen Komponenten implementiert werden. Oft liefern beide eine ähnliche Implementierung, manchmal aber auch komplett unterschiedliche Lösungen ein und desselben Problems. Durch die jeweils rein globale, nach unten gerichtete, bzw. rein lokale, nach oben gerichtete, Sichtweise kann es in beiden Fällen zu ineffizienter Programmierung kommen, weil man Zusammenhänge oder Ähnlichkeiten übersieht.

Bei mehreren Programmierern gibt es mehrere Möglichkeiten, die Aufgaben zu verteilen oder aufzuteilen. Typisch ist bei einem Programm, das nur eine Aufgabe erledigen soll, eine vertikale Aufteilung in verschiedene Teilaufgaben, z.B. kümmert sich eine Gruppe nur um die Benutzerschnittstelle, eine andere um die algorithmische Behandlung einer Menge von Teillösungen und eine dritte um die Bereitstellung dieser Teillösungen. Gibt es mehrere, von einander unabhängige Aufgaben, findet man häufig eine Aufteilung, die eine obere Lage und eine untere Lage hat (für Benutzerschnittstellen bzw. elementare Ein- und Ausgabe und gemeinsamen Strukturen) und dazwischen eine Aufteilung in mehrere Säulen aufweist.

Die Aufteilung ergibt sich in der Regel aus dem Datenfluss: Wo immer Teilaufgaben keine Daten austauschen, können sie unabhängig von einander implementiert werden. Wesentlich ist natürlich, dass man sich im Voraus oder aber möglichst frühzeitig darauf einigt, wer wem welche Daten übergibt und welche er zurückbekommt, was unbedingt nötig ist usw. Umgekehrt ist es sinnvoll, Module so zu gestalten, dass möglichst wenige wechselseitige Abhängigkeiten bestehen. Es gibt übrigens so etwas wie eine Faustregel für die Maximal-Anzahl an Funktionsparametern: Diese sollte höchstens fünf sein. Warum? Ansonsten ist die Funktion zu spezialisiert und vor allem ist ihre Bedienung zu unübersichtlich: Neue Parameter werden ohne nachzudenken separat an die Liste angehängt, obwohl sie vielleicht woanders noch mit hineinpassen würden, die Wartbarkeit und Überprüfbarkeit der Funktionen sinkt usw. Eine Möglichkeit ist die Zusammenfassung eines Teils der Parameter in Strukturen. Eine andere ist die Neuaufteilung des Problems, so dass Teilfunktionen entstehen, die mit weniger Parametern aufgerufen werden. Dem steht unter Umständen entgegen, dass man nicht unnötig Code verdoppeln will oder dass ein Teil der Parameterliste aus einer Struktur kommt, die man nicht als Ganzes übergeben will, um der Funktion genau die Daten zu geben, die sie braucht – und nicht mehr.

Um nun die Einzelteile schon während der Programmierphase testen zu können, gibt man sich in der Regel Testdaten vor und ersetzt Funktionen, die noch nicht geschrieben sind und logisch unter den aktuellen liegen, durch Platzhalter, die halbwegs brauchbare Werte zurückliefern bzw. nichts tun.

Hier kommt man fast automatisch zum Thema Testen. Eine einfach zu realisierende Art der Tests sind so genannte *unit tests*. Hierbei schreibt man sich zu jeder Funktion automatisch eine Testfunktion, die ein paar Standardwerte und ein paar Schlüsselwerte überprüft. Im Fall einer Funktion, der eine ganze Zahl übergeben wird, überprüft man Werte in dem Bereich, der einen interessiert, sowie 0, 1, $-1$ und die Extremwerte des Variablen-Wertebereichs. Gibt es andere logische Umschlagpunkte, wo vielleicht ein Teilbereich auf andere Weise behandelt wird als ein anderer, sollten die Werte unmittelbar davor und danach getestet werden usw. Die Testfunktionen einer Übersetzungseinheit (.c-Datei) werden dann von einer zentralen Testfunktion aufgerufen, die Datei-Testfunktionen wiederum von einer für das Modul oder die Bibliothek. So kann man in einfacher Weise nach Änderungen einen Gesamttest durchführen, und weiß später genau, wo Fehler entstehen (wenn man gute Testwerte verwendet hat).

Um fehlerfrei entwickeln zu können, sollte man in der Entwicklungsphase immer alle Warnungen

ansehen. Wenn man zu der Überzeugung gelangt, dass die Entwicklung eines Moduls abgeschlossen ist, kann man unnötige Warnungen durch entsprechende Flags in den Makefiles ausblenden – wenn man am Modul etwas ändert, sollte man dann allerdings wieder alle Warnungen ansehen.

Durch strukturiertes Vorgehen, sinnvolle Aufgabenteilung, frühzeitiges Testen und Dokumentieren kann man sich viel Arbeit sparen.

### 9.2.2. Umgebung optimieren

Wer mehr tut, als ab und zu Hundert-Zeilen-Programme zu basteln, sollte sich seine Arbeitsumgebung ansehen. Dinge, die einen ständig stören, sollten nicht zum Aufbau eines dauerhohen Frustlevels verwendet werden, sondern geändert werden. Wenn es einen stört, dass man sich ständig durch viele Fenster durchklicken muss zum Editieren, Kompilieren und Programmlauf anstoßen, dann sollte man sich nach einer integrierten Entwicklungsumgebung (*Integrated Development Environment*, IDE) umsehen oder nach einem Editor, der das u.U. auch kann. Hat man immer wiederkehrende Arbeitsschritte, dann sollte man diese automatisieren. Ein Editor mit eigener Makrosprache bietet die Möglichkeit, mit einem Tastendruck Code-Regionen zum Kommentar zu machen bzw. den Kommentar zu entfernen, bestimmte Ersetzungen vorzunehmen usw. Des weiteren arbeitet man ja nicht im luftleeren Raum, kann also auf vom Betriebssystem oder der Shell zur Verfügung gestellte Programme zugreifen. Unter Umständen lohnt es sich auch, einfach einmal bei bestimmten Problemen eine Suchmaschine zu bemühen – vielleicht hat ja schon jemand eine Lösung gebastelt, die man verwenden kann (hat man das selbst für bestimmte Dinge erledigt, sollte man natürlich umgekehrt auch so sozial sein, die eigenen Lösungen zur Verfügung zu stellen).

Durch die Verwendung geeigneter Werkzeuge wird das Erstellen von Dokumentation wie z.B. eigenen man-pages für die Funktionen und Datenstrukturen oder Tests stark vereinfacht.

Nicht zuletzt kann man die sogenannte *turn around*-Zeit für das Kompilieren, Linken und Laufenlassen manchmal durch geeignete Parameter verkürzen (oder dadurch, dass man nicht im Hintergrund tausend andere Programme am Laufen hat).

Wenn die Umgebung gut oder gar perfekt auf die eigenen Bedürfnisse abgestimmt ist, spart man sich langfristig sehr viel Zeit und Frust.

### 9.2.3. Code wiederverwenden

Nicht zuletzt kann man sich viel Zeit dadurch sparen, dass man immer wiederkehrende Aufgaben einmal sauber löst und dann immer die eigene Bibliothek verwenden kann. Typische Kandidaten sind Funktionen für das Bearbeiten und Auswerten der Kommandozeile, flexible Menüs, oft wiederkehrende Stringbearbeitungsfunktionen, reguläre Ausdrücke, Dateioperationen u.ä. Es ist oft sinnvoll, nicht-zeitkritische Programmteile gleich „bibliotheksfähig“ zu schreiben.

Unter Umständen hat man auch die Möglichkeit, wenn man ähnliche Aufgaben von vornherein sieht, bestimmte Aufgabentypen zusammenzufassen und von einer einzigen, etwas allgemeineren Funktion erledigen zu lassen.

Für größere Aufgaben lohnt sich auch eine Recherche, ob nicht jemand anders eine Bibliothek zur Verfügung stellt, die das Gewünschte leistet oder – falls der Quellcode offengelegt ist – entspre-

chend erweitert werden kann. Hier können Open-Source-Entwicklungen gegenüber proprietären Anwendungen Vorteile ausspielen.

Nicht zuletzt braucht man möglicherweise nicht eine Zeile in ***C*** zu programmieren, wenn die gewünschte Funktionalität in ausreichender Qualität und Geschwindigkeit von Shell-Kommandos oder mächtigen Skriptsprachen zur Verfügung gestellt wird und das in einer Art und Weise, dass die Erstellung eines entsprechenden Skriptes um ein Vielfaches schneller möglich ist.

## 9.3. Zeitaufwand optimieren (elementar)

> Premature optimization is the root of all evil.
>
> *(C.A.R. Hoare)*
>
> More computing sins are committed in the name of efficiency (without necessarily achieving it) than for any other single reason - including blind stupidity.
>
> *(W.A. Wulf)*

Bevor man sich mit der wichtigeren Optimierung der Laufzeit von Algorithmen beschäftigt, ist es sinnvoll, sich mit ein paar „grundlegenden Wahrheiten“ zu beschäftigen. Diese sind nicht in einem strikten Sinn wahr, d.h. sie führen nicht automatisch zu schnellerem Code. Außerdem kann man sich durch wohlgemeinte Optimierungsversuche durchaus den Code zerschießen, oder dafür sorgen, dass er nicht mehr portabel ist, dass er nur auf einer bestimmten Maschine schneller läuft oder gar nicht schneller läuft, oder dass der Compiler den Code nicht mehr optimieren kann. Nichtsdestotrotz kann die Beherzigung von einigen Grundregeln einiges oder zumindest ein Quentchen mehr an Geschwindigkeit bringen.

Die erste Grundregel hierbei lautet, dass man überprüfen sollte, ob eine Optimierung an der betrachteten Stelle insgesamt etwas bringt. Eine Verkürzung der Laufzeit einer Funktion um 50% bringt herzlich wenig, wenn die Funktion einen Anteil von einem Tausendstel Promille der Programmlaufzeit hat, verglichen mit einer Verkürzung um 3%, wenn man einen Anteil von zwei Drittel der Gesamtprogrammlaufzeit hat. Daher sollte man die zeitkritischen Programmteile identifizieren und vorrangig diese optimieren.

Als nächstes sollte man überprüfen, ob die Compiler-Optimierungen nicht schon ausreichen: Der Compiler „weiß“ meistens wesentlich besser als der Programmierer, welche Techniken auf dem entsprechenden Computer zu mehr Geschwindigkeit führen. Oft sorgen eigene Optimierungen dafür, dass der Compiler nicht mehr so gut optimieren kann und man letzten Endes sogar ein langsameres Programm erhält…

Es bringt nahezu nie etwas, in der Entwicklungsphase elementare Optimierungen vorzunehmen. Erst wenn das Programm fertig ist, insofern, dass es fehlerfrei läuft, ist die Frage, ob an der einen oder anderen Stelle mehr Geschwindigkeit vonnöten ist. Sollte schon bei allerersten Tests klar werden, dass man ein viel zu langsames Programm produzieren wird, ist es eher sinnvoll, sich auf die Suche nach einem besseren Algorithmus zu machen oder den Anforderungskatalog einzuschränken.

Im Folgenden stellen wir uns dennoch auf den Standpunkt dessen, der unbedingt jedes bisschen Geschwindigkeit braucht.

### 9.3.1. So spät wie möglich und so früh wie nötig

Einiges an Zeit kann gewonnen werden, indem man sich seine Schleifen vornimmt und überprüft, ob wirklich jedes Stück Code darin in die Schleife muss. Wenn man Zuweisungen darin vergessen hat, die logisch nicht jedes Mal erfolgen müssen, oder immer wieder die gleiche Bedingung abfragt, die nicht von den Laufvariablen der Schleife abhängt, besteht die einfachste Optimierung darin, diese Konstrukte nach außerhalb zu verlagern.

Während man im obigen Fall Dinge zu spät erledigt hat, ist es umgekehrt auch möglich, Dinge zu früh zu tun. ***C***99 bietet die Möglichkeit, Variablen an beliebiger Stelle zu deklarieren – für Optimierungszwecke sollte man das ausnutzen und die Variablen erst dann anlegen und vor allem erst dann initialisieren, wenn man sie wirklich braucht. Ein großes Array am Anfang einer Funktion mit Nullen zu beschreiben, wenn es nur in einem speziellen Fall überhaupt gebraucht wird, ist natürlich Zeitverschwendung.

Die Deklaration von Variablen zu verschieben bringt bei automatischen Variablen unter Umständen etwas Zeitersparnis. Zudem kann oft schon bei der Deklaration die Definition erfolgen, was mitunter auch Zeit spart.

### 9.3.2. Funktionen und Funktionsparameter

Ein Funktionsaufruf kostet Zeit für den Sprung an eine andere Stelle, aber vor allem durch die Übergabe der Parameter bzw. die Rückgabe des Ergebnisses, da diese in der Regel in Registern oder im Speicher abgelegt werden müssen. Bei potenziell großen Objekten wie Strukturen lohnt es sich daher nahezu immer, statt des Objektes lediglich den Speicherplatz des Objektes zu übergeben, ganz abgesehen von der Speicherersparnis.

Rekursionen sollten nur dort verwendet werden, wo sie die natürliche Weise sind, den Algorithmus auszudrücken, oder wo sicher ist, dass der Zusatzaufwand durch die Funktion die Laufzeit nicht großartig verändert, bei gleichzeitig wesentlich höherer Übersichtlichkeit.

### 9.3.3. Abfragen minimieren

#### Abfragen können Zeit kosten

Wenn während des Programmlaufes eine Abfrage im Code ist, muss die Bedingung ausgewertet werden, um zu wissen, welchem Pfad der Programmverlauf folgt, `if` oder `else`. Moderne Prozessoren haben sehr viele Anweisungen auf einmal in Bearbeitung; diese sind in einer sogenannten Pipeline und werden mit jedem Takt eine Stufe weiter bearbeitet. Kommt eine Verzweigung durch eine Abfrage, dann wird aufgrund einer Vorhersage entschieden, wo vermutlich weitergemacht wird. War die Vorhersage falsch, dann ist in der Regel der Inhalt der Pipeline ungültig. Die Pipeline muss sich neu füllen, einige Takte gehen verloren.

Deshalb ist es sinnvoll, unnötige Abfragen zu eliminieren und mitunter auch, sinnvolle Abfragen umzustellen.

### Loop unrolling

Häufig sind Abfragen in der Form von Schleifenfortsetzungsbedingungen zu finden. Für kurze Schleifen benötigt dann die Entscheidung, ob die Schleife abgebrochen oder fortgesetzt werden soll, einen erheblichen Teil der Gesamtzeit für die Schleife. Wenn die Schleife ohnehin eine feste Anzahl von Durchläufen umfasst, kann es sinnvoll sein, sie durch die Einzelanweisungen zu ersetzen, z.B.

```
for (int i=0; i<3; i++)
    array[i] = 0;
```

durch

```
array[2] = array[1] = array[0] = 0;
```

oder

```
array[0] = 0;
array[1] = 0;
array[2] = 0;
```

Ist die Schleife zu lang, kann das allerdings sehr unübersichtlich werden und zu Fehlern führen.

### DUFF's device

Ein Beispiel für partielles loop unrolling ist das schaurig-schöne DUFF*'s device*[1], das in etwas veränderter Form hier wiedergegeben ist, weil der Original-Code heute wegen der Funktionsdeklaration nicht mehr standardkonform ist. Gefragt war eine Funktion, die bestimmte Werte an eine bestimmte Adresse schreibt (der Reihe nach; mindestens ein Wert). Ausgehend von:

```
void send(register short *to, register short *from,
          register int count)
{
    do
        *to = *from++;
    while(--count>0);
}
```

gelangt man durch partielles loop unrolling zu einer Schleife, die beispielsweise `count/8` Durchläufe von acht Zuweisungen `*to = *from++;` vornimmt und eine Restschleife mit `count%8` Durchläufen von jeweils einer Zuweisung `*to = *from++;` hintenanstellt. Noch schneller geht es mittels:

[1] TOM DUFF, der „Erfinder" meinte dazu in der ersten Nachricht, in der er das anderen Leuten zeigte, dass er seinem Werk mit einer Mischung aus Stolz und Abscheu gegenüberstehe.

```
void send(register short *to, register short *from,
          register int count)
{
    register int n = (count+7)/8;
    switch (count%8){
        case 0: do{     *to = *from++;
        case 7:         *to = *from++;
        case 6:         *to = *from++;
        case 5:         *to = *from++;
        case 4:         *to = *from++;
        case 3:         *to = *from++;
        case 2:         *to = *from++;
        case 1:         *to = *from++;
                }while(--n>0);
    }
}
```

Das ist übrigens nicht die schnellste Möglichkeit, um größere Mengen von Daten zu kopieren und war auch nicht dafür gedacht (ansonsten hätten wir auch ein `*to++` im Code)!

### Rechnen statt fragen

Mitunter kann man Werte auch direkt ausrechnen, indem man Abfragen durch entsprechende Berechnungen ersetzt. Das setzt die Lesbarkeit des Codes entscheidend herab. Logische Und- und Oder-Operation können durch Multiplikation und Additionen simuliert werden (allerdings ohne die gleiche Abbruchcharakteristik!), ein logisches Nicht ist dagegen nicht einfach zu realisieren.

Beispiel: Wir wollen

```
int c = d;
....
if ( (a==1) )
    c *= 2;
else // a == 0
    c *= 4;
```

ersetzen und wissen sicher (!), dass `a` nur die Werte 0 und 1 annimmt. Dann könnten wir stattdessen auch

```
int c = d;
....
c *= 2*(2-a);
```

Wissen wir das nicht sicher, dann steckt hier eine potenzielle Fehlerquelle, die zumindest in einem Testmodus mittels `assert()` abgesichert werden sollte – womit zumindest der Testmodus erheblich langsamer ist als das Original. Zudem ist nicht sicher, ob diese Variante tatsächlich schneller ist.

Wenn man eine `switch`-Anweisung hat, deren Fälle aus einem zusammenhängenden Bereich ganzer Zahlen bestehen, ist es möglich, dass die Adresse, an der mit der Programmausführung weitergemacht wird, direkt berechnet werden kann, so dass weniger Zeit verloren geht als bei einer direkten Umsetzung einer äquivalenten `if-else if-...-else`-Konstruktion, wo Fälle, die weiter unten stehen, mehr Zeit kosten.

Wenn man eine Vielzahl von Fällen hat, die verstreut sind über einen großen Zahlenbereich, ist es manchmal möglich, eine Abbildung zu finden, die die Einzelfälle injektiv in einen kleineren Zahlenbereich abbildet. Ist der Bild-Bereich dann beinahe zusammenhängend, ist es möglich, dass die `switch`-Anweisung trotz des Zusatzaufwandes im Durchschnitt schneller abgearbeitet wird. Diese Technik nennt sich *hashing*. Ist die hash-Abbildung surjektiv, so spricht man von einem perfekten hash. Es gibt Algorithmen und auch frei verfügbare Programme zu diesem Zweck.

**Hinweis:** Hashing kann auch in anderen Zusammenhängen Sinn machen, zum Beispiel zur Speicheroptimierung, wenn man für Schlüssel aus einem bestimmten zusammenhängenden Bereich Speicher braucht, die Schlüssel aber nur verstreut liegen. Umgekehrt kann man sich Zeit sparen, wenn man nur Speicher entsprechend der Schlüsselanzahl alloziert und dann normalerweise jedesmal alle Schlüssel ansehen müsste, um einen bestimmten zu finden; die hash-Abbildung ermöglicht dann einen direkten Zugriff mit konstantem Zeitaufwand.

Ähnliche Tricks sind das Aufspalten einer komplexen Vergleichsoperation, die sich nicht mit `switch` abhandeln lässt, in eine `switch`-Abfrage mit nachgeschalteter komplexer Mehrfachalternative. Beispiele wären das Abbilden einer Fließkommazahl auf einen Ganzzahl-Datentyp, bevor man in die Überprüfung auf feinere Intervalle einsteigt oder das Überprüfen des ersten Zeichens bei String-Vergleichen:

```
if (strcmp(str,"alle")==0) {
    ....
} else if (....) {
    ....
} ....
.... else if (strcmp(str,"mops")==0) {
    ....
} else if (strcmp(str,"moral")==0) {
    ....
} else if (strcmp(str,"morbid")==0) {
    ....
} else if (....) {
    ....
} ....
.... else if (strcmp(str,"zu")==0) {
    ....
} else {
    ....
}
```

wird dann zu

```
switch (str[0]) {
 case 'a':
    if (strcmp(str,"alle")==0) {
```

```
        ....
    } else if (....) {
        ....
    } ....
    else {
        goto MyDefault;
    }
    break;
 case 'b':
    ....

 case 'm':
    ....
    .... else if (strcmp(str,"mops")==0) {
        ....
    } else if (strcmp(str,"moral")==0) {
        ....
    } else if (strcmp(str,"morbid")==0) {
        ....
    } else if (....) {
        ....
    } ....
    ....

 case 'z':
    ....
    .... else if (strcmp(str,"zu")==0) {
        ....
    } else {
        goto MyDefault;
    }
    break;
 default:
MyDefault:
    ....
}
```

Anstatt mit der Funktion `strcmp()` Zeichen für Zeichen zu überprüfen und jedes Mal den Zusatzaufwand für den Funktionsaufruf in Kauf zu nehmen, wird sozusagen schon einmal grob vorsortiert, so dass nur noch wenige Aufrufe von `strcmp()` nötig sind. Die `goto`-Anweisung wird im nächsten Abschnitt erklärt.

### 9.3.4. `goto`

Eine ***C***-Anweisung ist bisher übergangen worden: `goto`. Sie gilt als pures Gift für strukturiertes Programmieren und kann seltsame Nebeneffekte hervorrufen. Die Syntax lautet

`goto` *Sprungmarke*;

wobei *Sprungmarke* sich in der gleichen Funktion befinden muss und wie die `switch`-Labels durch einen Doppelpunkt gekennzeichnet ist:

*Sprungmarke: Anweisung;*

Die `goto`-Anweisung sorgt für einen Sprung von der aktuellen Stelle zur ersten Anweisung nach der Sprungmarke.

In der Regel trägt `goto` nur zu unübersichtlichem Programmverhalten bei, aber es gibt einen Fall, in dem es nicht schneller als mit `goto` geht: Abbruch einer mehrfach geschachtelten Schleife. Mit unseren bisherigen Mitteln könnten wir das am besten durch einen Abbruchtest in jeder Schleife tun:

```
bool Abbruch = false;
while (1) {
    ....
    for (i=0; i<sehrgross; i++) {
        ....
        if (irgendwas(i)) {
            ....
            for (j=0; j<auchnichtklein; j++) {
                ....
                if (bedingung(i,j)) {
                    Abbruch = true;
                    break;
                }
            }
            if (Abbruch)
                break;
        }
    }
    if (Abbruch)
        break;
}
```

Das kostet in jedem Schleifendurchlauf Zeit für den Abbruchtest. Mit `goto` kann man sich die äußeren Tests sparen:

```
while (1) {
    ....
    for (i=0; i<sehrgross; i++) {
        ....
        if (irgendwas(i)) {
            ....
            for (j=0; j<auchnichtklein; j++) {
                ....
                if (bedingung(i,j)) {
                    goto Schleifenende;
                }
            }
        }
    }
}
```

```
Schleifenende:
```

Das ist allerdings der einzige Fall, in dem man `goto` verwenden sollte.

### 9.3.5. Code wiederverwenden?

Funktionsaufrufe kosten Zeit - dennoch sollte man nicht unbedingt auf Funktionen verzichten. Einen Teil der Code-Wiederverwendung kann man durch Makros und - wichtiger noch - `inline`-Funktionen erledigen, die beide keinen Overhead liefern (sollten).

Außerdem gibt es Bibliotheksfunktionen wie z.B. `memcpy()`, die das, was sie machen, gut und vor allem so schnell wie möglich auf dem jeweiligen Rechner machen. Diese Funktionen sollte man auf jeden Fall verwenden. Unter Umständen hat man auch selbst optimierte Versionen bestimmter Funktionen erzeugt, die sollte man auch wiederverwenden.

Mit Hilfe von bedingter Kompilierung ist es zudem möglich, ***C***-Code maschinenabhängig zu schreiben, ohne dass er nicht mehr portabel ist.

Dennoch sollte man nicht gedankenlos alle Funktionen der Standardbibliothek oder anderer Bibliotheken verwenden: Gerade Funktionen, die große Flexibilität gestatten und nicht auf einen bestimmten Datentyp festgelegt sind, haben einen höheren Verwaltungsaufwand, z.B. `bsearch()` oder `qsort()`. Wenn man diese Funktionen speziell für die eigenen Datentypen neu implementiert und dabei keinen Fehler macht, gelangt man zu einer wesentlich schnelleren Variante.

## 9.4. Speicheraufwand optimieren (elementar)

Oft ist nicht die Laufzeit eines Programmes das Problem, sondern die Beschränktheit des Speichers. Wenn man die Compiler-Optimierungen verwendet, sollte man deshalb darauf achten, dass man nur diejenigen verwendet, die nicht Geschwindigkeit auf Kosten eines höheren Speicheraufwandes erkaufen!

Speicher einzusparen ist eine Sache, die man generell von Anfang an tun sollte, allerdings gibt es Spezialitäten, die man sich nicht immer antun möchte.

Wie beim Zeitaufwand gilt auch hier: Erst herausfinden, was Speicher frisst. In der Regel sind das dynamisch allokierte Datentypen, die mit wachsender Problemgröße hinzukommen. Der Hauptaufwand sollte daher darauf gerichtet sein, solche Strukturen „kleinzukriegen“.

### 9.4.1. Nichts verdoppeln

Klingt einfach, ist mitunter aber schwierig. Wenn man so lange wie möglich keine globalen Variablen verwenden will (was extrem sinnvoll ist), dann ist manchmal die Verdopplung von Informationen nicht zu vermeiden.

Klassisches Beispiel einer eingesparten Verdopplung ist die Übergabe einer Adresse statt einer Variable (allerdings kosten auch Zeiger Speicherplatz). Bei Arbeiten auf großen Datenstrukturen erliegt man für manche Arbeiten, bei denen der Inhalt erheblich verändert wird, schnell der Versuchung, die Datenstruktur zu verdoppeln, anstatt nur die benötigten Teile zu verdoppeln.

### 9.4.2. Richtigen Typ verwenden

Die Auswahl des richtigen Datentyps kann auch entscheidend zur Ersparnis beitragen. Wenn eine `short`-Variable ausreicht, muss man nicht unbedingt ein `long long` verwenden, wenn `float` ausreicht, nicht unbedingt ein `double`.

Dank der Typdefinitionen in `<stdint.h>` kann man die Größe von Ganzzahl-Datentypen unter *C*99 auch gezielt auswählen.

Zusätzlich sei darauf verwiesen, dass auch die Reihenfolge der Einträge einer Struktur einen Einfluss auf die Gesamtgröße der Struktur hat, siehe 4.18.

### 9.4.3. Bit-Schubsereien

Im Prinzip das Gleiche noch einmal: Hat man viele Optionsvariablen, aber jeweils wenige Optionen, vielleicht auch nur die Wahl zwischen 0 und 1, so ist es sinnvoll, diese Optionen nicht in separaten Variablen zu speichern, sondern in ein oder zwei Variablen, und dann die Information in den Bits zu verpacken. Bitfelder sind unter Umständen eine bequemere Variante, bei ihnen ist allerdings nicht sichergestellt, dass der Speicherplatz gut ausgenutzt wird.

## 9.5. Bessere Algorithmen bauen

Am meisten Ersparnis bringt oft die Verwendung „besserer“ Algorithmen.

Um die Güte von Algorithmen messbar zu machen, sind formale Kriterien notwendig. Diese werden im Folgenden eingeführt und am Beispiel erklärt.

### 9.5.1. Komplexität: Operationen und Speicheraufwand zählen

Wie wir gesehen haben, kosten verschiedene Operationen unterschiedlich viel Zeit, deshalb ist es sinnvoll, diese Operationen separat zu zählen, um sicherzustellen, dass der Zeitaufwand richtig abgeschätzt werden können.

Gezählt werden in der Regel

- Abfragen, wobei man unter Umständen die Abfragen am Ende bzw. Anfang von Schleifen unterschlägt,
- Vergleiche,
- Additionen/Subtraktionen, wobei wiederum das Schleifeninkrement unter den Tisch fallen kann,
- Multiplikationen,
- Divisionen,

- Zuweisungen, wobei unterschiedliche kombinierte Zuweisungen unterschiedlich lange dauern können,
- bitweise Operationen,
- Zugriffe auf Werte in komplexen Datenstrukturen und Feldern,
- Aufrufe von *black-box*-Funktionen.

Früher haben Additionen und Subtraktionen wesentlich weniger Zeit gekostet als Multiplikationen oder Divisionen, mittlerweile verschwimmen oder verschwinden die Grenzen.

Für numerische Algorithmen werden oft jeweils eine Multiplikation oder Division kombiniert mit einer Addition oder Subtraktion als „Operation“ gezählt und mitunter der Rest an Operationen unterschlagen.

Für Aufgaben wie das Suchen oder Sortieren spielen Vergleiche, Zugriffe und Zuweisungen eine wesentliche Rolle. Abfragen sind meistens von einem Vergleich begleitet, weswegen man sie in der Regel zusammenfasst.

#### Individuelles Erbsenzählen

Ohne ausführliches Profiling und Tests kann man nicht sicher sein, was im verwendeten System die Dauer einer Operation ist, aber als Faustregel kann man sagen, dass bei Ganzzahl-Datentypen die durchschnittliche Dauer einer Zuweisung, einer arithmetischen oder bitweisen Operation, einer mit einer arithmetischen oder bitweisen Operation kombinierten Zuweisung, eines Feldzugriffs sowie eines Zugriffs auf einen elementaren Datentyp in einer Struktur oder Vereinigung ungefähr gleich sind und als Basiseinheit gelten können. Multiplikation, Division und Divisionsrestberechnung können u.U. aber bei Ganzzahl-Datentypen, die größer sind als `int`, auch länger dauern.

Fließkomma-Operationen dauern mindestens gleich lang wie ihre Ganzzahl-Pendants, u.U. für `long double` auch länger, hier sollte man auf jeden Fall Multiplikation und Division mit einem etwas höheren mittleren Aufwand veranschlagen.

Die mittlere Dauer eines Vergleichs ist in der Regel die einer einfache Operation, aber es ist sinnvoll, Vergleiche nur im Zusammenhang mit Abfragen zu sehen. Abfragen dauern – in Abhängigkeit von der Güte der Sprungvorhersage des Prozessors – im Mittel meist länger als andere Operationen.

Funktionen der Standardbibliothek, die Operanden fester Größe aufweisen, z.B. alle mathematischen Funktionen, können in dieses Schema als black-box-Operatoren eingebettet werden.

Ein- und Ausgabeoperationen auf Konsolen- oder Dateiebene sollten bei Möglichkeit außen vor gelassen werden, nur, wenn es algorithmisch erforderlich ist – etwa beim Bearbeiten von Objekten, die nicht in den Hauptspeicher passen – sollte man sie im Algorithmus belassen und mitzählen.

Bei Funktionen mit Operanden variabler Größe muss eine mittlere Dauer in Abhängigkeit der Operandengrößen herausgefunden werden. Hier ist es oftmals sinnvoller, eine eigene Variante der Funktion zu programmieren, deren Verhalten man kennt und deren elementare Operationen man zählen kann.

Hier betrachten wir jetzt zwei Beispiele, an denen wir uns die Operationen und Operationenzahlen klarmachen können:

**`strchr()`:** Die Funktion `strchr()` ermittelt das erste Vorkommen eines gegebenen Zeichens in einem gegebenen String und liefert einen Zeiger auf das Zeichen bei Erfolg und `NULL` bei Misserfolg (also Nichtvorkommen des Zeichens). Eine Implementierung könnte etwa folgendermaßen aussehen:

```
char *strchr(const char *s, int c)
{
    while (*s != (char) c)
        if ( (*s++) ==’\0’ )
            return NULL;
    return s;
}
```

Ist `c` an der $n$-ten Position, so testen wir die Schleifenfortsetzungsbedingung $n$-mal, und die innere Bedingung $(n-1)$-mal. Ist die Länge des Strings inklusive Terminator $N$ und `c` kommt kein einziges Mal vor, so testen wir beide Bedingungen $N$-mal. Nachdem Inkrementoperationen meistens nichts zusätzlich kosten, können wir unsere Kosten als $2n-1$ (bzw. maximal $2N$) *Abfrage+Vergleich*-Operationen angeben.

**`mysqrt()`:** Die nichtrekursive Variante der HERONschen Regel aus dem letzten Kapitel soll abgezählt werden. Wir nehmen an, wir kennen eine Funktion $a(\cdot)$, so dass für den von uns berechneten Startwert die Iterationszahl für eine Zahl $N$ für die Erreichung der gewünschten Genauigkeit bei $a(N)$ liegt[2]. Das Anlegen von Variablen wird nicht gezählt, die Rückgabe ebenfalls nicht. Jetzt können wir anfangen zu zählen:

```
1    double approx;
2
3    if(base<=0.0)                                  1 Abfrage+Vergleich
4        return 0.0;
5
6    approx = (1.0+base)/2.0;                       1 Zuweisung, 1 MulAdd
7    do {                                           ab hier alles mal a(base)
8        approx = 0.5*(approx+base/approx);         1 Zuweisung, 2 MulAdd
9    } while (ABS((approx*approx/base-1.0))>PRECVAL);
                                    1 ABS mal 2 MulAdd, 1 Abfrage+Vergleich
A
B    return approx;
```

Die Kosten für `ABS()` sind

```
C    #define ABS(I) ((I)>0 ? (I) : -(I))
                                    1 Abfrage+Vergleich, 2 Evaluationen,
```

[2]Eine obere Schranke kann man beispielsweise durch die Betrachtung von $N = 10, 10^2, 10^3, \ldots$ gewinnen, indem man die Anzahl der Schritte zählt, bis die erste Stelle stimmt (eine Möglichkeit ist $0.55 \cdot N^{0.28}$ aufgerundet für $N > 1$); danach verdoppelt sich die Anzahl der gültigen Stellen in jedem Schritt, für 15 gültige Stellen braucht man dann noch zusätzliche 4 Schritte.

ergibt also $2a(base)+1$ mal *Abfrage+Vergleich*, $a(base)+1$ mal *Zuweisung*, $4a(base)+1$ mal *MulAdd*.
Durch das vorgeschlagene teilweise loop unrolling (sechsmal das Innere der Schleife) kommen wir auf $2a^*(base)+1$ mal *Abfrage+Vergleich*, $6a^*(base)+1$ mal *Zuweisung*, $14a^*(base)+1$ mal *MulAdd*, wobei $a^*(base)$ sich als $a(base)/6$ (aufgerundet) ergibt. Schätzen wir das durch das größere $1+a(base)/6$ ab, wird der Gewinn deutlicher: Wir gelangen zu $a(base)/3+2$ mal *Abfrage+Vergleich*, $a(base)+2$ mal *Zuweisung*, $(2+1/3)\,a(base)+2$ mal *MulAdd*; bis auf die Zuweisungen, die die billigste Operation sind, geht die Anzahl der Operationen überall herab, wenn wir $a(base)\geq 2$ annehmen.

### LANDAU-Symbole und Komplexität

In beiden Beispielen lassen sich die Maximal-Kosten durch die Abhängigkeit von $N$ bzw. *base* ausdrücken. Im zweiten Fall haben wir lediglich konstante Vorfaktoren verbessert, die Bibliotheksfunktion `sqrt()` hat demgegenüber aber keine Abhängigkeit von *base*, also konstanten Aufwand – zumindest für große *base* ist `sqrt()` also „besser“ als `sqrt()`.

An dieser Stelle lohnt es sich, entsprechende Begriffe einzuführen; der Vollständigkeit halber werden in der folgenden Definition die LANDAUschen Symbole in einem allgemeinen Kontext definiert.

**Definition:** *Sei $M\subset\mathbb{C}$ und sei $a\in\mathbb{C}\cup\{\infty\}$ ein Häufungspunkt von $M$, ferner $f,g:M\to\mathbb{C}$ und $r:M\to\mathbb{R}_+$. Man schreibt mit den* LANDAU*schen Symbolen $\mathcal{O}$, $o$:*

$$f(z)=g(z)+\mathcal{O}(r(z)) \text{ für } z\in M :\Leftrightarrow \exists c\in\mathbb{R}_+\forall z\in M:\frac{|f(z)-g(z)|}{r(z)}\leq c$$

$$f(z)=g(z)+o(r(z)) \text{ für } z\in M, z\to a :\Leftrightarrow \lim_{M\ni z\to a}\frac{f(z)-g(z)}{r(z)}=0$$

Für unsere Zwecke können wir uns auf $M=\mathbb{N}$ und $a=\infty$ beschränken; im Folgenden sind diese Werte immer implizit angenommen. Zudem gehen wir von $z$ auf $n$ über.

**Bemerkungen:**

1. Die LANDAU-Symbole arbeiten mit Funktionenklassen. Gilt $f(n)=g(n)+\mathcal{O}(r(n))$, so gilt auch $f(n)=g(n)+\mathcal{O}(C\cdot r(n))$ für beliebiges $C\in\mathbb{R}_+$. Das Gleichheitszeichen ist hier *keine* Äquivalenzrelation mehr, sondern bezeichnet das Enthaltensein in einer Klasse, die durch $g(n)+\mathcal{O}(r(n))$ ausgedrückt werden kann, d.h. alles, was einmal im Argument von $\mathcal{O}$ steht, kann rechts vom nächsten Gleichheitszeichen nicht wieder herausgenommen werden und eine Aussage der Form $\mathcal{O}(r(n))=f(n)$ ist schlechterdings Unsinn.

2. Offensichtlich gilt $o(r(n))=\mathcal{O}(r(n))$.

3. Für ein Polynom $p$ vom Grad $k$ gilt $p(n)=\mathcal{O}(n^k)=\mathcal{O}(n^{k+l})$, $l\in\mathbb{R}_+$.

4. Wir werden die LANDAU-Symbole dazu verwenden, um den asymptotischen Speicher- und Zeitaufwand von Algorithmen zu charakterisieren. Kennt man ein scharfes (oder hinreichend scharfes) Argument $f(n)$ für $\mathcal{O}$, so heißt $f(n)$ *Komplexität* (*Zeitkomplexität*, *Speicherkomplexität*) des Algorithmus. Scharf meint hierbei, dass für alle Funktionen $g$ mit

$g(n)/f(n) \to 0$ für $n \to \infty$ gilt, dass Zeit- bzw. Speicheraufwand nicht $\mathcal{O}(g(n))$ sind.
Oft genügt auch eine nicht ganz scharfe Eingrenzung, um die Güte des Algorithmus einordnen zu können.

5. Bei der Betrachtung der Komplexität lässt man offensichtlich alle Konstanten unter den Tisch fallen, weswegen die Betrachtung der Komplexität allein nicht notwendigerweise zum geeignetsten Algorithmus führt[3].

### 9.5.2. Situation evaluieren

Um einen effizienten Algorithmus entwerfen oder einen bestehenden Algorithmus verbessern oder anpassen zu können, sollte man im Voraus wissen, in welchem Umfeld er verwendet wird.

Die Beschränkung auf einen bestimmten Typ von Eingabeparametern oder die Verwendung von Zusatzinformationen kann eine Variante ermöglichen, die wesentlich schneller ist. Genauso ist es möglich, dass, wenn nur eine Teilinformation der eigentlichen Ausgabe verwendet wird, das Liefern dieser Teilinformation mit einem billigeren Algorithmus zu haben ist.

Als Beispiel soll uns das Finden eines Vierecks in einem Netz mit viereckigen Maschen zu einem bestimmten Punkt dienen. Jede Masche ist konvex und hat Informationen über ihre bis zu vier Nachbarn sowie einen logischen Nachfolger, siehe Abbildung 9.1. Berandet der Rand des Maschennetzes ein Gebiet, wie hier der Fall, so nennen wir die Maschen Elemente. Die Elemente kennen die Knoten, durch die sie definiert sind, und die Knoten wiederum kennen ihre Koordinaten. Wir können für ein konvexes Polygon mit einheitlichem Umlaufsinn der Ecken (hier: Gegenuhrzeigersinn) leicht herausfinden, ob ein Punkt $p$ innerhalb oder außerhalb des Elementes liegt, indem wir für alle Seiten testen, ob der Punkt auf der Innenseite liegt, in unserem Fall bei der Seite $i$=$(i,(i+1)\%4)$ links. Das lässt sich dadurch überprüfen, dass wir das innere Produkt der äußeren Normalen von $i$ mit der Strecke $(i,p)$ überprüfen. Der Knoten mit der lokalen Nummer $i$ im Element habe den globalen Index $I$ und der Knoten $i+1$ den globalen Index $J$, dann müssten wir also

$$c_{\alpha_{i,p}} = (-y_J + y_I)(x_p - x_I) + (x_J - x_I)(y_p - y_I)$$

berechnen. Ist $c_{\alpha_{i,p}} > 0$, so liegt $p$ außerhalb, ansonsten müssen wir weitertesten, siehe auch Abbildung 9.2. Punkte auf einer Elementseite ($c_{\alpha_{i,p}} = 0$) werden als innerhalb gewertet, weil sie ansonsten keine Zuordnung fänden und man nicht feststellen könnte, ob sie überhaupt im Gebiet oder auf dem Rand des Gebietes liegen. Damit gelangen wir für $N$ Elemente zu einem Aufwand von $(1Z + 4(5A + 2M + 1T))N$ im schlimmsten Fall, wobei $Z$ der Aufwand für die Zuweisung eines Zeigerwertes (nächstes Element), $A$ der Aufwand für eine Addition, $M$ der Aufwand für eine Multiplikation, $T$ der Aufwand für den Test, d.h. den Vergleich plus die Abfrage. Im Mittel muss man nur die Hälfte der Elemente testen und weiß auch im Mittel bei den Nichttreffern nach der Hälfte der Seiten, dass $p$ nicht im Element liegt, mithin gelangt man also im Mittel zu einem Aufwand von $(1Z + 2(5A + 2M + 1T))N/2$

Liegt der Durchmesser aller Elemente im Bereich um $h$, dann kann die Elementzahl mit einer geeeigneten Konstante $C$ durch $N \approx Ch^{-2}$ abgeschätzt werden. Wird $h$ halbiert durch ein feineres

[3] Ein schönes Beispiel hat C. Bau in comp.lang.c gegeben: *„Consider method A taking $N^{\frac{2}{3}} \log^2(N)$ nanoseconds, and method B taking $N^{0.7}$ nanoseconds. For large N, method A is faster. For any problem that can be solved in a billion years, method A is slower.“*

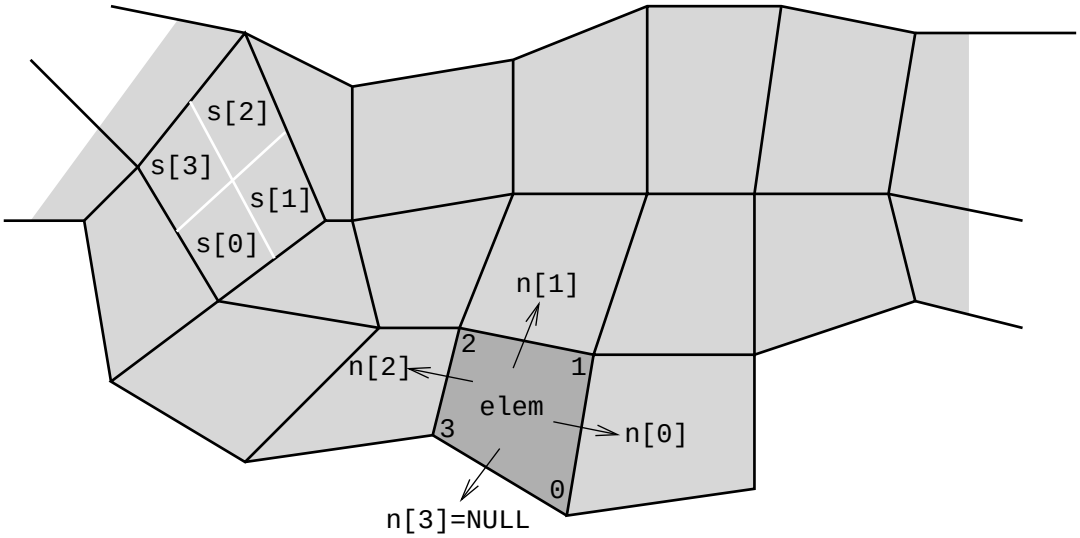



**Abbildung 9.1.:** Das unterlegte Gebiet wird in viereckige Maschen aufgeteilt, sogenannte Elemente. Ein Element wird durch vier Knoten beschrieben, deren lokale Nummerierung von 0 bis 3 läuft. Die Seiten werden von den Knoten aus nummeriert, Seite $i$ läuft von Knoten $i$ nach $(i+1)\%4$. Zeiger auf die Nachbarelementen sind in dem Feld $n[]$ gespeichert, der Feldindex leitet sich von der Seite ab, über die der Nachbar zu erreichen ist. Randseiten führen zu keinem Nachbarn.

Netz, so vervierfacht sich die Zahl der Elemente, also auch der Aufwand für das Testen aller Elemente, halbiert man $h$ noch einmal, liegt man beim sechzehnfachen Aufwand usw.

Nutzt man allerdings die Tatsache aus, dass das feinste Gitter durch diese Art der Verfeinerung aus dem gröbsten Gitter hervorgegangen ist, kann man den Zeitaufwand wieder verringern, indem man diese Zusatzinformation abspeichert: Man geht vom gröbsten Gitter aus und speichert in jedem Element auch Verweise auf die vier Sohn-Elemente, die durch einmaliges Verfeinern aus dem Element hervorgehen. Dabei ist der Sohn $s[i]$ dasjenige Sohn-Element, das den lokalen Knoten $i$ des Vater-Elements enthält. Die Sohn-Elemente verweisen wiederum auf ihre Söhne usw. (Abbildung 9.3) bis hin zum feinsten Gitter. Will man nun nach dem Element suchen, das den Punkt $p$ enthält, so fängt man auf dem gröbsten Gitter an und sucht das grobe Element, das den Punkt enthält, sucht von diesem ausgehend auf dem nächstfeineren Gitter dasjenige Sohnelement, das den Knoten enthält, usw. bis zum feinsten Gitter. Wir bezeichnen mit $L$ die Anzahl der Verfeinerungen, dann ist der Maximalaufwand $(1Z+4(5A+2M+1T))(N+4L)$ anstelle von $(1Z+4(5A+2M+1T))N^{2L}$. Nutzt man die Tatsache aus, dass für je zwei Seiten eines Sohnelements (nämlich für $s[i]$ die Seiten $i$ und $(i+3)\%4$) schon bekannt ist, dass $p$ nicht außerhalb liegt, reduziert sich der Maximalaufwand auf $(1Z+4(5A+2M+1T))N+(1Z+2(5A+2M+1T))4L)$, bedenkt man, dass das Vaterelement konvex ist und durch zwei Geraden in die vier Sohnelemente aufgeteilt werden kann, so genügt es, jeweils nur die Seiten 1 und 2 von $s[0]$ zu überprüfen: Liegt $p$ außerhalb bezüglich 1, so liegt $p$ entweder in $s[1]$ oder in $s[2]$, ansonsten liegt $p$ in $s[0]$ oder in $s[3]$. Der Test bezüglich Seite 2 liefert bei außerhalb entweder $s[3]$ oder $s[2]$ und bei innerhalb entweder $s[0]$ oder $s[1]$, also ist eindeutig bestimmt, in welchem Sohn $p$ liegt. Staffelt man das entsprechend, so gelangt man zu $(1Z+4(5A+2M+1T))N+(1Z+2(5A+2M+1T))L)$. Der Preis an Speicher, den man zahlt ist, dass man anstelle von $N$ Elementen $N\sum_{\nu=0}^{L}(1/4)^{\nu}$ speichern muss und jedes Element um vier Verweise anwächst, d.h. für die Elemente hat man nun einen

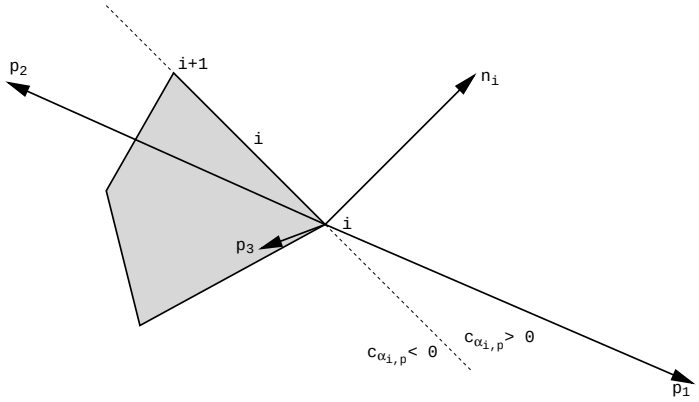



**Abbildung 9.2.:** Das innere Produkt des äußeren Normalenvektors $n_i$ der Seite $i = (i, i+1)$ mit dem Vektor $(i, p_k)$ gibt Auskunft darüber, ob der Punkt $p_k$ sicher außerhalb liegt, wie im Fall $p_1$, aber nicht darüber, ob ein Punkt innerhalb liegt wie im Fall $p_3$ oder eben außerhalb bezüglich einer anderen Seite, hier zum Beispiel $((i+2)\%4, (i+3)\%4)$.

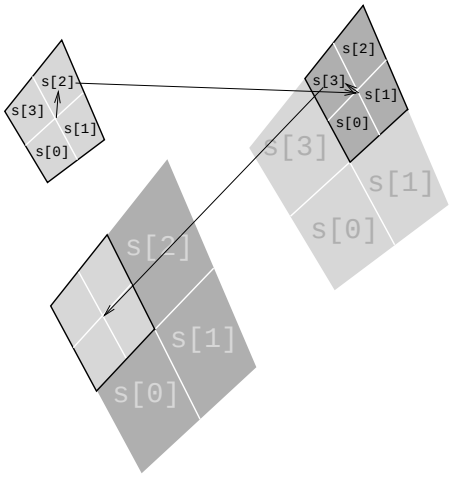



**Abbildung 9.3.:** Abstieg aus dem Element aus Abbildung 9.1, dessen Sohn-Elemente bereits dargestellt waren, zum Sohn $s[2]$ und von dort wiederum zu dessen Sohn $s[3]$.

Speicheraufwand von weniger als $(4/3)(1+4(\texttt{sizeof Verweis})/(\texttt{sizeof Elem\_alt}))$ zusätzlich (der Vorfaktor $4/3$ stammt aus der geometrischen Reihe und wird nie erreicht). Dazu kommen beim Zeitaufwand noch die Zeit für die Zuweisung der Sohn-Referenzen bei der Gitterverfeinerung ($(4^{L+1}-1)ZN/3$).

Hier hat uns die Konvexität einen entscheidenden Vorteil beim Suchen gebracht. Wissen wir, dass das Gebiet selbst konvex ist, können wir die Suche auf dem gröbsten Gitter ebenfalls abkürzen: Wann immer $p$ außerhalb bezüglich einer Seite liegt, fahren wir nicht mit dem nächsten Element fort, sondern mit dem Nachbarn bezüglich dieser Seite. Ist unser Startelement in der Mitte des Gebiets, so müssen wir nur den halben Gebietsdurchmesser zurücklegen. Sind die Winkel der Elemente nach unten beschränkt, können wir berechnen, wieviele Elemente wir höchstens zu durchlaufen haben (hier geht auch die kleinste Element-Seitenlänge oder bei entsprechenden unteren und oberen Winkelbedingungen der minimale Elementdurchmesser mit ein). Auf jeden Fall lässt sich diese Zahl als $Ch^{-1}$ schreiben, man gelangt also von $\mathcal{O}(N)=\mathcal{O}(h^{-2})$ zu $\mathcal{O}(h^{-1})$.

Noch viel schneller und einfacher geht es, wenn unsere Elemente in einem regelmäßigen Gitter angeordnet sind, also achsparallel bezüglich der Achsen eines Koordinatensystems sind. Dann lässt sich für konvexe Gebiete bei entsprechender Elementsortierung direkt aus den Koordinaten von $p$ das Element berechnen, in dem $p$ liegen muss: Gehen wir von quadratischen Elementen aus mit Seitenlänge $s=h/\sqrt{2}$ und $n_x$ Elementen in $x$- und $n_y$ Elementen in $y$-Richtung, wobei o.B.d.A. die linke untere Ecke der Ursprung sein soll. Wir geben dem ersten Element (am Ursprung) die Nummer 0 und wählen als Nachfolger jeweils das nächste Element in positiver $x$-Richtung. Das Element $k\cdot n_x-1$ (am Ende einer solchen Reihe) hat als Nachfolger dann jeweils das von Element 0 ausgehend $k$-te Element in positiver $y$-Richtung. Damit ergibt sich die Elementnummer als $e:=n_x\lfloor p_y/s\rfloor+\lfloor p_x/s\rfloor$, wobei $\lfloor z\rfloor$ die Nachkommastellen von $z$ streicht, also abrundet. Jetzt muss man nur noch bis zum Element $e$ laufen. Das dauert im Durchschnitt $N/2$ Zuweisungen. Effizienter ist es, ein Feld $P$ der Größe $N=n_xn_y$ mit Verweisen auf die Elemente anzulegen, weil man dann einfach mit konstantem Aufwand auf $P[e]$ zugreifen kann, mithin haben wir also einen Aufwand von $1Z+1A+2T+2M$ für das Auffinden des Elements, bei einem zusätzlichen Speicheraufwand von $N(\texttt{sizeof Verweis})$ und einem Initialisierungsaufwand für das Sortieren (Größenordnung zwischen $\mathcal{O}(N\log N)$ und $\mathcal{O}(N^2)$, siehe Abschnitt 9.5.3). Besser geht es nicht. Für ein nichtkonvexes Gebiet greift das allerdings ins Leere.

Dennoch gibt uns das eine Idee für den allgemeinen Fall, wie man entweder zusammen mit der Verfeinerungsstrategie oder direkt auf dem feinsten Gitter suchen kann. Wir überdecken das Gebiet mit einem Rechteck und unterteilen das Rechteck selbst wieder in $n_x\times n_y$ Teilrechtecke, sogenannte Boxen. Zu jeder Position im Rechteck können wir wie oben erklärt, die Box finden, das diese Position enthält. „Weiß“ die Box, welche Elemente einen nichtleeren Schnitt mit ihm haben, so kann man durch eine verkürzte Suchliste hindurchlaufen. Ein einfacher Algorithmus, der das bewerkstelligt, könnte so funktionieren: In einem ersten Durchlauf durch alle Knoten finden wir die linke untere und rechte obere Ecke eines solchen Rechtecks heraus. Um Ärger mit Rundungsfehlern zu vermeiden, ist es sinnvoll, die Länge und Breite des Rechtecks um den Faktor $1+2\varepsilon$, $\varepsilon>0$ klein, zu strecken. Dann alloziert man anhand der gewünschten Boxgröße oder anhand der gewünschten Feinheit in $x$- und $y$-Richtung Platz für die Boxen. Nun kann man einmal durch alle Elemente durchlaufen, berechnet für jedes Element die linke untere und rechte obere Ecke eines Rechtecks, das es überdeckt, und trägt das Element in alle Boxen ein, in denen das überdeckende Rechteck Punkte hat, siehe auch Abbildung 9.4. Auf diese Weise überschätzt man allerdings die Anzahl der Elemente pro Box für bestimmte Fälle. Wenn Speicher das begrenzende Element

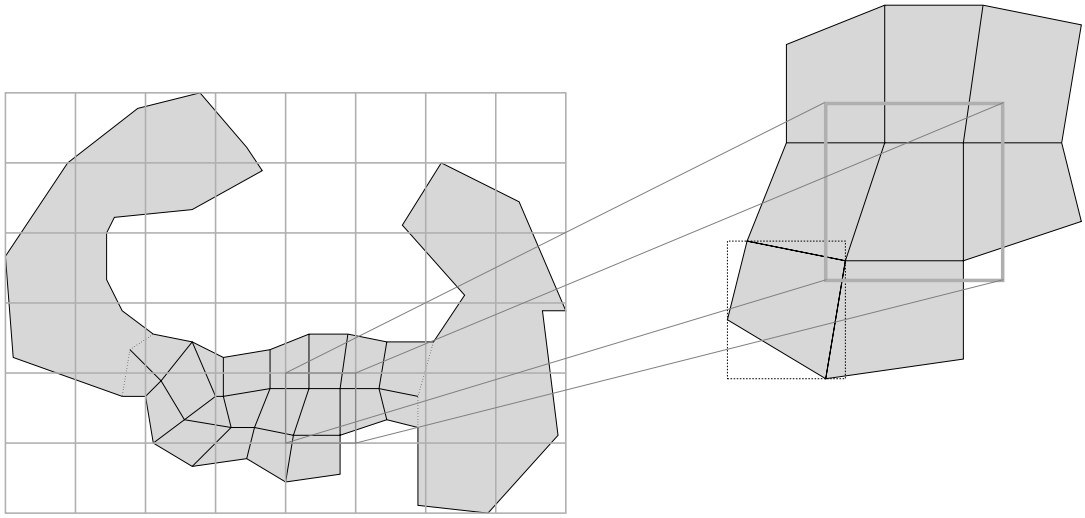


**Abbildung 9.4.:** Überdeckung des (ergänzten) Gebietes aus Abbildung 9.1 durch ein Rechteck und quadratische Boxen (links) und Vergrößerung einer Box mit den enthaltenen Elementen (rechts). Zu dem Element links unten ist das überdeckende Rechteck dargestellt, dessen rechte obere Ecke in der Box liegt, so dass das Element in die Box-Elementliste eingetragen wird.

ist, kann man für alle interessanten Boxen überprüfen, ob auch wirklich mindestens einer ihrer Eckpunkte in dem Element liegt. Sinnvoller ist es dann allerdings, die Initialisierung der Boxen mit zwei Durchläufen durch alle Elemente zu erledigen: Im ersten Durchlauf speichert man nur ab, wieviele Elemente in einer Box liegen, im zweiten Durchlauf alloziert man die entsprechende Speichermenge und trägt die Elemente ein, um sich den Aufwand für eine Liste zu sparen. Nun stellt sich nur noch die Frage, wie groß die Boxen zu wählen sind, ob man die Boxenmethode mit einer Suche über mehrere Gitterebenen kombinieren will, usw. Für die Größe der Boxen gilt es zu bedenken, dass, wenn das Rechteck, das das Gebiet überdeckt, eine wesentlich größere Fläche hat als das Gebiet selbst, es sehr viele leere Boxen gibt. Es ist sinnvoll, die Boxengröße in einem Bereich zwischen „jedes Element liegt in höchstens vier Boxen, jede Box hat die achtfache Fläche des größten Elements“ und „jede Box hat die Fläche des kleinsten Elements“ zu wählen.

Bei weiteren Informationen kann man entsprechend schnellere oder speicherschonendere Algorithmen entwickeln.

### 9.5.3. Beispiel: Sortieren

Abschließend wollen wir uns mit einer Aufgabe und zugehörigen Algorithmen, die häufiger vorkommt und intuitiv leicht verständlich ist: Sortieren.

Zunächst sehen wir uns ein einfaches Beispiel an, wo der Mensch selbst sortiert, um zu verstehen, welche Möglichkeiten man hat, und unter welchen Voraussetzungen man wie agiert. Im weiteren Verlauf der Analyse werden wir dann sehen, dass der Mensch die Lösung der Aufgabe für seine Zwecke eigentlich schon ganz gut optimiert hat – und dass künstlich erzeugte, einfach zu programmierende Algorithmen nicht unbedingt besser sein müssen.

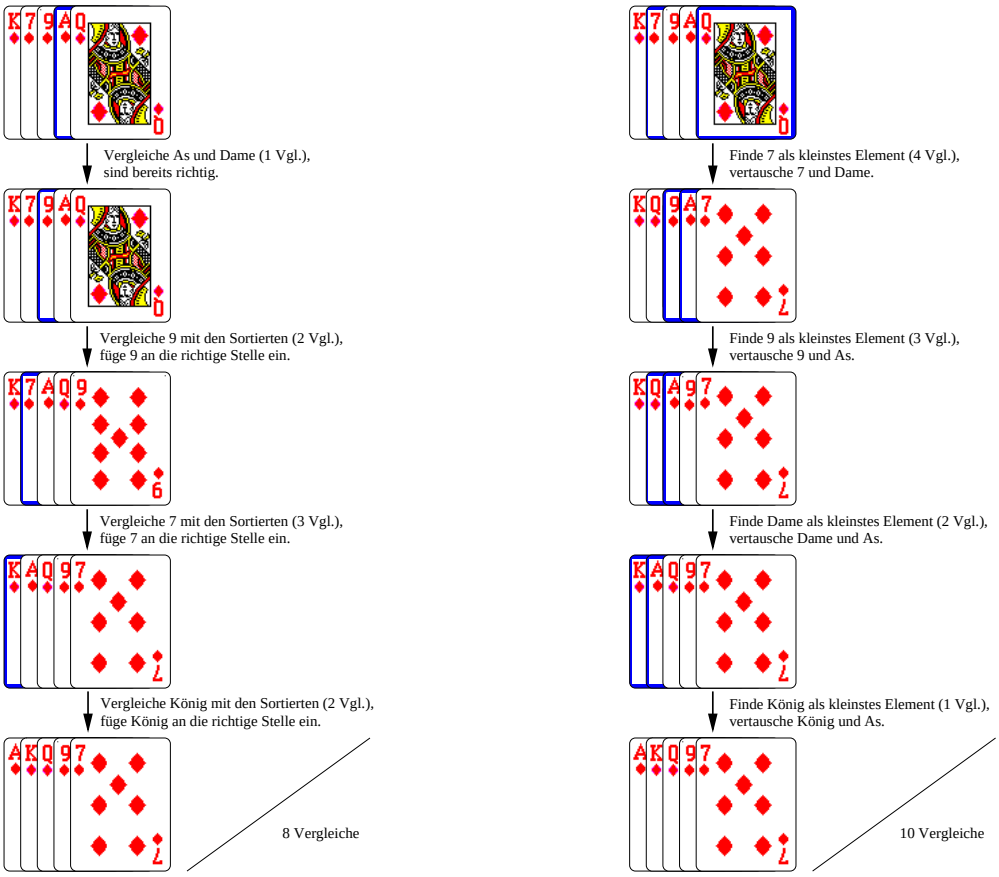



**Abbildung 9.5.:** Links: Sortieren durch Einfügen; nicht richtig einsortierte Karten werden an die richtige Stelle gebracht. Rechts: Sortieren durch Auswählen; man bringt die kleinste Karte nach vorn, die zweitkleinste an die zweite Stelle usw.; im Gegensatz zum normalen Vorgehen wird hier aber nur ein Auschtausch von zwei Karten vorgenommen.

### Intuitives Verhalten analysieren

Das menschliche Gehirn ist bereits ein äußerst leistungsfähigerer (lokaler) Optimierer. Deswegen lohnt es sich bei Aufgaben, die im menschlichen Bereich vorkommen, sich zu überlegen, wie man es selbst machen würde – und sich nicht von vornherein den Blick durch die tatsächlich verwendeten Datenstrukturen zu verstellen. Das kann man dann mit dem vergleichen, was sich ob der Datenstrukturen anbietet.

Im Beispiel mit der Elementsuche lässt sich das Vorgehen im letzten Fall damit vergleichen, dass man weiß, wie das Gebiet ungefähr liegt und deshalb weiß, auf welche Elemente man die Suche beschränken kann.

Für das Sortieren schauen wir uns an, was man bei Kartenspielen macht, wenn man Karten auf die Hand bekommt, und sie erst einmal sortiert, um sich einen Überblick zu verschaffen (wenn man kein semiprofessioneller Zocker ist, der das auch ohne Sortieren kann. . . ).

- Eine Möglichkeit ist, den Kartenstapel Karte für Karte aufzudrehen und jede Karte, die nicht passt, an der richtigen Stelle einzuordnen (*insertion sort*);

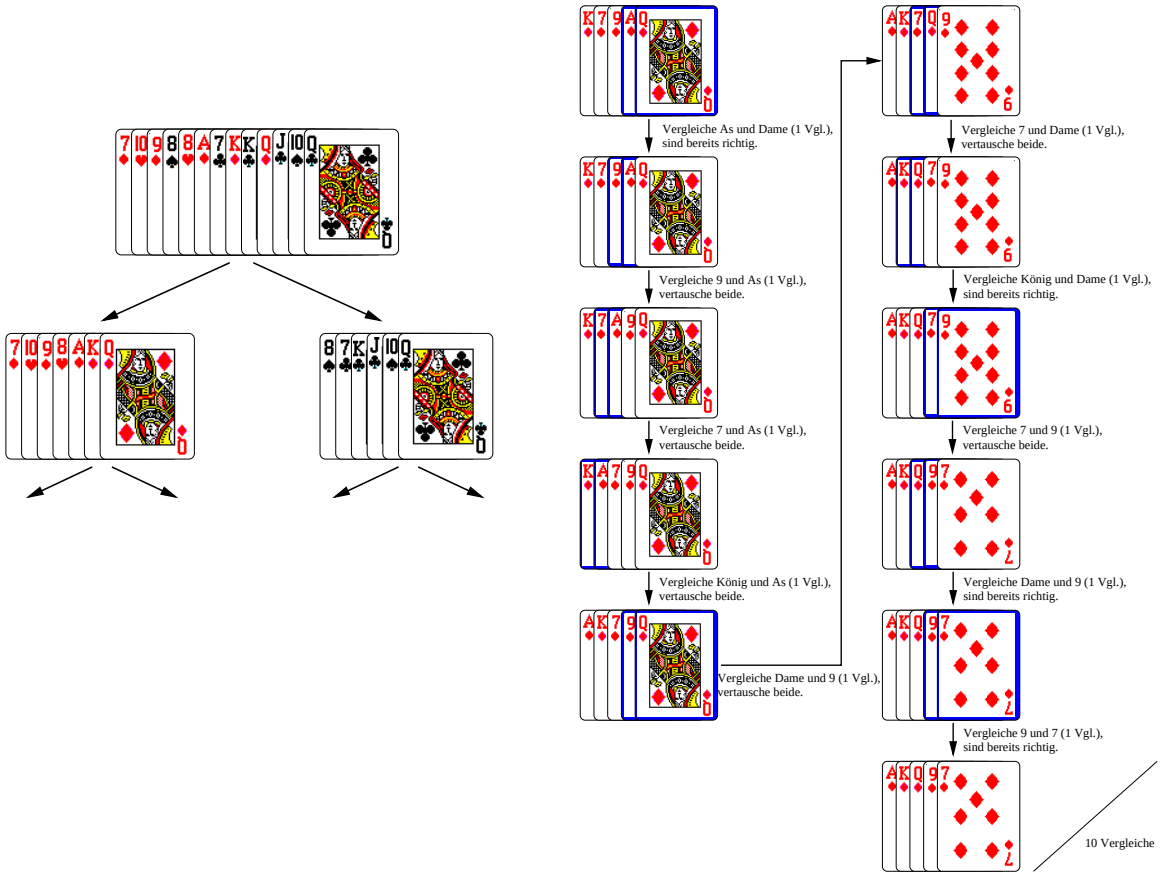



**Abbildung 9.6.:** Links: Divide et impera; man kann die Karten nicht in der Hand halten und sortieren, also splittet man den Kartenstapel so lange nach gewissen Kriterien, bis man die Teilstapel sortieren kann. Rechts: Sortieren durch lokales Herstellen der richtigen Ordnung; ist man einmal durch, so ist das größte Element hinten, beim zweiten Durchlauf ist das zweitgrößte Element an vorletzter Stelle.

- eine andere ist, sich die (bei nur einer Kartenfarbe bzw. für die Kartenfarben einzeln) kleinste Karte zu suchen und einzufügen, dann die zweitkleinste usw. (nicht unähnlich dem *selection sort*);

- bei vielen Karten und zwei schwarzen und zwei roten Kartenfarben könnte man auch auf die Idee kommen, erst einmal alle schwarzen Karten beiseite zu legen, dann die roten nach Herz und Karo aufzuteilen und dann erst Herz nach einer der beiden vorigen Methoden zu sortieren, dann Karo, und anschließend Schwarz in Pik und Kreuz aufzuteilen usw. (in gewisser Weise Vorstufe zum *quick sort*);

- hat man mehrere Helfer, könnte man auch den Stapel so aufteilen, dass jeder Helfer eine kleine, durch eines der ersten beiden sortierbare Menge an Karten bekommt, diese sortiert, dann jeweils zwei Leute ihre beiden Stapel zu einem größeren vereinen, indem einer seine Karten beim anderen an der richtigen Stelle einordnet, bis man zum Schluss wieder einen (sortierten) Stapel hat (eine Art *merge sort*).

Die Abbildungen 9.5 und 9.6 (links) zeigen die ersten drei Möglichkeiten überblicksweise.

Aufgabe ist es, ein Feld von `int`-Werten der Länge *n* zu sortieren. Wir betrachten zunächst elementare Sortieralgorithmen, die eine Zeitkomplexität $\mathcal{O}(N^2)$ aufweisen:

Die erste Variante, die direkt dem Bild beim Kartensortieren entspricht, ist das Verfahren *insertion sort*.

***Programmlisting 9.1:*** sorting/insertionsort.c

```
#include <stdlib.h>

int insertionsort (int *array, size_t number)
{
    for (size_t i=1; i<number; i++) {
        size_t j = i;
        int save = array[i];
        while (j>0 && array[j-1] > save) {
            array[j] = array[j-1];
            j--;
        }
        // "Dreiecks"tausch abschliessen
        array[j] = save;
    }

    return 0;
}
```

Es kann eventuell effizienter gemacht werden, indem man nicht alle unpassenden Werte `int`-weise verschiebt, sondern mit Hilfe der Funktion `memmove()` den gesamten Bereich auf einmal verschiebt.

Beim *selection sort* fügt man nicht die kleinste Karte vorne ein, sondern vertauscht die kleinste, zweitkleinste, usw. Karte mit der ersten, zweiten, usw. Karte, so dass man nur einen Tausch hat und nicht viele.

***Programmlisting 9.2:*** sorting/selectionsort.c

```
#include <stdlib.h>

int selectionsort (int *array, size_t number)
{
    for (size_t i=0; i<number-1; i++) {
        size_t min=i;
        for (size_t j=i+1; j<number; j++) {
            if (array[j] < array[min])
                min = j;
        }
        // Dreieckstausch
        int t = array[min]; array[min] = array[i]; array[i] = t;
    }

    return 0;
}
```

Es gibt einen weiteren elementaren Sortieralgorithmus, der für Programmierkurse sehr beliebt ist, weil er sich so einfach implementieren lässt: *bubble sort*. Hierbei vergleicht man jeweils zwei benachbarte Elemente und sortiert sie richtig. Am Ende des ersten Durchlaufs ist entweder das größte oder das kleinste Element an der richtigen Stelle (je nach Durchlaufrichtung), das dann im nächsten Durchlauf ausgelassen werden kann usw. bis nur noch zwei Elemente übrig sind. Diese werden ggf. vertauscht – und das Feld ist richtig sortiert.

***Programmlisting 9.3:*** sorting/bubblesort.c

```
#include <stdlib.h>

int bubblesort (int *array, size_t number)
{
    for (size_t i=number; i>0; i--) {
        for (size_t j=1; j<i; j++) {
            if (array[j-1] > array[j]) {
                // Dreieckstausch
                int t = array[j-1]; array[j-1] = array[j]; array[j] = t;
            }
        }
    }

    return 0;
}
```

Überlegt man sich das mit den Karten, so kommt einem die Sache unnötig umständlich vor – zu Recht.

Um nun die Qualität der Algorithmen beurteilen zu können, kann man sich überlegen, wie sie sich im Hinblick auf Vergleiche und Vertauschungsoperationen schlagen. Hierzu ist es sinnvoll, sich drei Fälle anzusehen: Feld ist bereits sortiert, Feld ist genau falsch herum sortiert, Feld ist zufällig sortiert und man geht von mittleren Suchzeiten aus.

Darüber sollte man allerdings ein gezieltes Profiling nicht vergessen.

Der bubble-sort-Algorithmus ist am einfachsten zu betrachten: Wir haben zwei geschachtelte Schleifen, die Abfrage im Inneren wird jedes Mal durchgeführt, die Vertauschung nur, wenn der Vergleich positiv ausfällt. Wir haben also auf jeden Fall $n(n-1)/2$ Abfragen und im günstigsten Fall 0, im ungünstigsten Fall $n(n-1)/2$ und im Durchschnittsfall um die $n(n-1)/4$ Dreieckstäusche.

Beim insertion sort hat man im günstigsten Fall $n-1$ Vergleiche und doppelt so viele Zuweisungen von `int`-Werten. Im ungünstigsten Fall hat man $n(n-1)/2$ Vergleiche, im Durchschnitt etwa $n^2/4$ Vergleiche und $n(n+1)/2-1$ bzw. etwa $n^2/4$ Zuweisungen, man liegt also meist bei weniger Vergleichen und Zuweisungsoperationen als bubble sort.

Der dritte Algorithmus, selection sort, gibt uns immer $n(n-1)/2$ Vergleiche und immer $n-1$ Dreieckstäusche sowie zwischen 0 und $n(n-1)/2$ Zuweisungen für den Index.

Für fast sortierte Felder ist somit insertion sort am besten geeignet, genauso im Durchschnittsfall für billige Zuweisungen. Ist dagegen die Zuweisung eine teure Operation, weil beispielsweise nicht `int`-Werte sortiert werden sollen, sondern die `int`-Werte Sortierschlüssel von Strukturen darstellen, dann ist selection sort am besten geeignet. Wenn die Menge der möglichen Schlüssel-

werte kleiner und das Auswerten der Abfragen billiger wird, wird der Vorteil des selection sort immer deutlicher.

Im Buch von Sedgewick [Sed92] wird etwas anders gezählt.

Alle drei betrachteten Verfahren sind für das Sortieren von elementaren Datentypen von der Zeitkomplexität $\mathcal{O}(n^2)$ mit verschiedenen Konstanten aber alle mit wenig zusätzlichem Speicheraufwand.

Wird statt eines Feldes eine andere Datenstruktur verwendet, z.B. eine Liste, ist es meistens sinnvoller, Zeiger auf die Listenelemente in einem Feld abzuspeichern, dieses Feld zu sortieren und hinterher die Liste neu zu verketten, weil der wahlfreie Zugriff auf das $i$-te Listenelement auch $i$ Zugriffsoperationen bedeutet, während beim Feld nur die zum $i$-ten Element gehörige Adresse berechnet wird und dann auf die Adresse zugegriffen wird. Die mittlere Zugriffszeit ist also bei der Liste $n/2$ Zugriffsoperationen, mithin $\mathcal{O}(n)$, beim Feld $\mathcal{O}(1)$. Andererseits lässt sich insertion sort auch sehr einfach auf verkettete Listen umschreiben, das Verschieben des Blocks entspricht zwei Verkettungsoperationen. Hebt man ausreichend Zeiger auf, entsteht bis auf Konstanten kein zusätzlicher Aufwand, für das Verschieben des Blocks kommt man von $\mathcal{O}(n)$ auf $\mathcal{O}(1)$.

### Shell sort

Insertion sort ist im ungünstigsten Fall deswegen so langsam, weil falsch plazierte Elemente nur sehr langsam an den richtigen Platz im Feld geschafft werden.

Shell sort führt insertion-sort-Läufe für gleichabständige Feldelemente durch, wobei ein kleinerer Abstand, der nicht 1 ist, offensichtlich kein Teiler des nächstgrößeren sein sollte. Man fängt mit dem größten Abstand an und sortiert sich dann nach unten bis zum Abstand 1. Weil insertion sort für fast sortierte Felder in etwa lineare Zeitkomplexität hat, wird das Verfahren sozusagen immer schneller.

***Programmlisting 9.4:*** sorting/shellsort.c

```
#include <stdlib.h>

int shellsort (int *array, size_t number)
{
    size_t h;
    for (h=1; h<number/9; h = 3*h +1);
    for ( ; h>0; h/=3) {
        for (size_t i=h; i<number; i++) {
            size_t j = i;
            int save = array[j];
            while ((j >= h) && (array[j-h]>save)) {
                array[j] = array[j-h];
                j -= h;
            }
            array[j] = save;
        }
    }

    return 0;
}
```

Man kann für die Abstandsfolge $1, 4, 13, 40, 121, \ldots$ ($d_i = 3\,d_{i-1} + 1$) beweisen, dass Shell sort nie mehr als $n^{\frac{3}{2}}$ Vergleiche ausführt.

Die Zeitkomplexität des Shell sort ist nicht bekannt, wird aber zwischen $\mathcal{O}(n(\log n)^2)$ und $\mathcal{O}(n^{1.25})$ vermutet.

### Quick sort

Der Algorithmus quick sort ist ebenfalls ein gutes Allzweck-Verfahren. Die Idee ist rekursiv am einfachsten zu erklären:

1. Finde guten Schätzwert für den Median der Schlüssel (das ist der Wert, der bei sortiertem Feld im mittleren bzw. einem der mittleren Elemente steht).
2. Ordne alle kleineren Schlüsselwerte links, alle größeren rechts davon an.
3. Sortiere die linke Teilliste mit quick sort.
4. Sortiere die rechte Teilliste mit quick sort.
5. Füge die Teillisten zusammen und kehre zurück.

Wenn wir den Median hätten, würde bei jedem Aufruf des quick sort die Liste in zwei gleich große Teile gespalten, die ihrerseits mit qick sort sortiert werden, d.h. auch wenn man in jeder Rekursionsebene alle Elemente anschaut, so gibt es doch nur größenordnungsmäßig $log_2(n)$ Rekursionsebenen, der Zeitaufwand ist $\mathcal{O}(n \log n)$.

Hier liegt natürlich die crux: Um den Median zu bestimmen, müssen wir das Feld sortieren... Wenn man stattdessen ein beliebiges Element wählt, erwischt man im ungünstigsten Fall jeweils ein maximales oder minimales Element und eine der Teillisten ist jeweils leer, man hat $n$ Rekursionsebenen, quadratische Zeitkomplexität (bei schlechteren Konstanten als die elementaren Verfahren).

Eine Möglichkeit zur Vermeidung bzw. zum Senken der Wahrscheinlichkeit des ungünstigsten Falles ist, drei Elemente zu wählen, deren Median zu bestimmen, und diesen Elementwert zu verwenden.

Zudem ist für kleine (Teil-)Feldgrößen die Verwendung eines elementaren Sortierverfahren wegen der kleineren Konstanten günstiger.

Eine Beispielimplementierung könnte etwa folgendermaßen aussehen:

***Programmlisting 9.5:*** sorting/quicksort.c

```
#include <stdlib.h>
#include "insertionsort.h"

#define MY_THRESH 6

int quicksort_r (int *array, size_t l, size_t r)
{
    if (r-l>=MY_THRESH) {
```

```
        // Aufteilelement ermitteln
        int divide[] = { array[l], array[(l+r)/2], array[r] };
        insertionsort(divide, 3);
        // Mittleres Element an die Stelle r-1
        array[l] = divide[0]; array[r] = divide[2];
        array[(l+r)/2] = array[r-1];
        int t, save = array[r-1] = divide[1];
        size_t i=l, j=r-1;
        // bezueglich save aufteilen
        while (1) {
            // naechstes falsches von unten bzw oben ermitteln
            while (array[++i] < save);
            while (array[--j] > save);
            if (i>=j)
                break;
            // falsch sortierte Elemente tauschen
            t = array[i]; array[i] = array[j]; array[j] = t;
        }
        // save richtig einsortieren
        t = array[i]; array[i] = save; array[r-1] = t;
        // Teilprobleme loesen
        quicksort_r(array,l, i-1);
        quicksort_r(array,i+1, r);
    }
    else { // kleines Teilproblem -> insertion sort ist schneller
        insertionsort(&array[l], r-l+1);
    }

    return 0;
}
```

Die Funktion `quicksort` wird im Header-File über die rekursive Variante gestülpt, um die gleiche Schnittstelle zu bieten.

Wie wir bereits festgestellt haben, kosten Rekursionen einiges an Zeit (und Speicher). Es ist möglich, die Rekursion im quick sort zu beheben, indem man Start- und Endindex der noch zu bearbeitenden Teilfelder auf einem Stapel ablegt. Wird immer das größere Teilfeld abgelegt, so muss nicht in jedem Durchlauf der Schleife dynamisch ein neues Stapelelement alloziert werden, sondern der Stapel kann am Anfang groß genug alloziert werden…

Quick sort hat einen deutlich größeren Zusatz-Speicheraufwand als die anderen betrachteten Verfahren.

### Spezialfall: Bucket sort

Für sehr kleine Schlüsselgrößen gibt es ein Verfahren von linearer Komplexität. *Bucket sort* (*bucket* = Eimer) arbeitet relativ einfach: Man hat für jeden Schlüssel einen Eimer und wirft die Schlüssel einfach in den passenden Eimer, d.h. man hat nur einen Durchlauf durch das Feld. Hinterher werden alle Eimer der Reihe nach angesehen und die Schlüssel werden entsprechend ihrer Vielfachheit an der richtigen Stelle zurückgeliefert.

In Tabelle 9.1 findet sich der Ablauf für das Beispiel `EIN_SORTIERBEISPIEL`. Die Buckets sind in diesem Fall die Schlüssel A-Z sowie _ (statt Leerzeichen). Wenn die Schlüssel nicht gleich

| | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z | _ |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| E | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| I | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| N | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| _ | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| S | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| O | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| R | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| T | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| I | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 2 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| E | 0 | 0 | 0 | 0 | 2 | 0 | 0 | 0 | 2 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| R | 0 | 0 | 0 | 0 | 2 | 0 | 0 | 0 | 2 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 2 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| B | 0 | 1 | 0 | 0 | 2 | 0 | 0 | 0 | 2 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 2 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| E | 0 | 0 | 0 | 0 | 3 | 0 | 0 | 0 | 2 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 2 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| I | 0 | 0 | 0 | 0 | 3 | 0 | 0 | 0 | 3 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 2 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| S | 0 | 0 | 0 | 0 | 3 | 0 | 0 | 0 | 3 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 2 | 2 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| P | 0 | 0 | 0 | 0 | 3 | 0 | 0 | 0 | 3 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 2 | 2 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| I | 0 | 0 | 0 | 0 | 3 | 0 | 0 | 0 | 4 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 2 | 2 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| E | 0 | 0 | 0 | 0 | 4 | 0 | 0 | 0 | 4 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 2 | 2 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| L | 0 | 0 | 0 | 0 | 4 | 0 | 0 | 0 | 4 | 0 | 0 | 1 | 0 | 1 | 1 | 1 | 0 | 2 | 2 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |

**Tabelle 9.1.:** Bucket sort für `EIN_SORTIERBEISPIEL`.

den zu sortierenden Elementen sind, ist jeder Eimer der Start einer einfach verketteten Liste, in die die Elemente mit entsprechendem Schlüssel eingefügt werden.

Der entscheidende Nachteil des bucket sort ist der enorme Speicheraufwand für mittlere bzw. große Schlüsselräume.

Eine Abart des bucket sort ist das *lexikographische Sortieren*. Hierbei geht man von gleich oder ähnlich langen Schlüsseln aus, die aus wenigen Stellen bestehen, die ihrerseits als Schlüssel für den bucket sort interpretiert werden können; wir gehen von Schlüsseln der Länge $m$ mit einem Stellenraum der Größe $l$ aus (bucket sort: Speicheraufwand $\mathcal{O}(l^m)$, Zeitaufwand $\mathcal{O}(n)$): Man sortiert die Schlüssel nach der letzten Stelle mittels bucket sort, die so sortierten Schlüssel werden dann nach der vorletzten Stelle sortiert mittels bucket sort usw. So erspart man sich den Speicheraufwand des bucket sort (hier Speicheraufwand $\mathcal{O}(l)$ und Zeitaufwand $\mathcal{O}(mn)$).

## Beispielprogramme

Ein Beispiel-Hauptprogramm und die Header für die Sortierverfahren.

***Programmlisting 9.6:*** sorting/main.c

```
/*****************************************************************
**
**  sorting
**
*****************************************************************/

/*** INCLUDES & DEFINES ******************************************/

#include <stdio.h>      // fprintf, fopen, fclose
#include <stdlib.h>     // malloc, free, exit, rand
#include <string.h>     // strtoll, strchr, memcpy
#include <time.h>       // time (for srand)
#include <limits.h>     // CHAR_BIT
#include <assert.h>     // assert

/* sorting algorithms */
#include "selectionsort.h"
#include "insertionsort.h"
#include "bubblesort.h"
#include "shellsort.h"
#include "quicksort.h"

#define NAMLEN         80
#define TESTNUMSTRING  "20"


/*** TYPEDEFS & GLOBAL VARIABLES *********************************/

/* Usage: Enter new sorting algorithm in list of algos after creating
** an enum entry.
*/

/* Enumeration of possible sorting algos */
typedef enum sortingalgos {START=-1,
    selection,
    insertion,
    bubble,
    shell,
    quick,
    LAST
} Counter;

/* Function pointer type for sorting algos */
typedef int (* MySortProcPtr) (int *, size_t);

/* Structure containing all algo information */
typedef struct {
    char algochar;           // I/O identifier
    char *algoname;          // Name to be printed
    MySortProcPtr algofunc;  // Function pointer
```

```
} SortingInfo;

/* Actual list; global for convenience */
static SortingInfo MyInfo[] = {
    [selection] = {'s', "selection sort"   , selectionsort},
    [insertion] = {'i', "insertion sort"   , insertionsort},
    [bubble]    = {'b', "bubble sort"      , bubblesort},
    [shell]     = {'S', "Shell sort"       , shellsort},
    [quick]     = {'q', "quick sort"
                        " (recursive,"
                        " middle of three,"
                        " insertion sort)" , quicksort}
};


/*** FUNCTION PROTOTYPES ****************************************/

static void goodbye (const char *pname);
static unsigned getmodes (const char *modstring);
static int *getfield (const char *namestring, size_t *size);



/*****************************************************************
**
** main program
**
** Arguments: see goodbye()
**
** Job:       Acquire an array of integers, sort them according to the
**            required algorithms, give estimate of execution time.
**
** Returns:   0            .. on success
**  exits     EXIT_FAILURE .. memory allocation failed
**            termination by other functions possible
*/

int main (int argc, char **argv)
{
    unsigned int modes;
    size_t num = 0;
    int *T = NULL, *testval = NULL;
    clock_t myclock;
    double mytime;
    FILE *outptr = stdout;

    if (argc>1) {
        if (argv[1][0]!='-') {
            goodbye(argv[0]);
        }
        modes = getmodes(argv[1]);
        if (argc>2) {
            T = getfield(argv[2], &num);
        }
        else
            T = getfield(TESTNUMSTRING, &num);
```

```
        if (T==NULL)
            goodbye(argv[0]);

        testval = malloc(num*sizeof(int));
        if (testval==NULL) {
            free(T);
            fprintf(stderr, "%s, line %d: Memory allocation failed."
                            " Aborting...\n", __func__,
                            (int) __LINE__);
            exit(EXIT_FAILURE);
        }
    }
    else
        goodbye(argv[0]);

    for (size_t i=0; i<num; i++)
        fprintf(outptr,"%d, ",T[i]);
    fputs("\n",outptr);

    for (Counter mycnt=START+1; mycnt<LAST; mycnt++) {
        if (modes & (1<<mycnt)) {
            memcpy(testval, T, num*sizeof(int));
            myclock = clock();
            if((*(MyInfo[mycnt].algofunc))(testval, num))
                fputs("Fehler!\n", stderr);
            myclock = clock()-myclock;
            mytime = (double)myclock/CLOCKS_PER_SEC;
            fprintf(outptr, "Algorithm: %s.  Elapsed time: %8.6g\n",
                            MyInfo[mycnt].algoname, mytime);
            for (size_t i=0; i<num; i++)
                fprintf(outptr, "%d, ",testval[i]);
            fputs("\n", outptr);
        }
    }

    return 0;
}


/*********************************************************************
**
** goodbye - throw out the command line syntax and exit
**
*/

static void goodbye (const char *pname)
{
    fprintf(stderr, "\nUsage: %s -<mode> [file|num]\n\n * mode:\n",
                    pname);
    for (Counter i=START+1; i<LAST; i++)
        fprintf(stderr, "   \t%c .. %s\n",
                        MyInfo[i].algochar, MyInfo[i].algoname);
    fprintf(stderr, " * file:\n"
                    "   \tProcesses a file with white space separated"
                        " integers\n");
```

```
    fprintf(stderr, " * num:\n"
                    "   \tGenerates num (default %s) random "
                                    "integers\n",
                    TESTNUMSTRING);
    exit(EXIT_SUCCESS);
}


/*********************************************************************
**
** getmodes - find out which algos are wanted and return corresponding
**            modes; die on too many possible modes
**
*/

static unsigned getmodes (const char *modstring)
{
    unsigned modes = 0;

    /* safety check: We do not have more algos than bits in type of
    ** modes. If this fails, proceed to larger type or array. */
    assert (sizeof(modes)*CHAR_BIT>=LAST);

    for (Counter i=START+1; i<LAST; i++) {
        if(strchr(modstring,MyInfo[i].algochar))
            modes |= 1<<i;
    }

    return modes;
}


/*********************************************************************
**
** getfield - acquire array of integers
**
** Arguments: namestring .. string containing a filename or number
**            size       .. size_t * to store the size of the acquired
**                           array
**
** Job:       Test for only a number in the string, if yes, generate
**            *size = this number random integers. If no, try to open
**            the content of the string as file and read in the
**            integers there. Supported file format: see goodbye()
**
** Returns:   NULL         .. Failure
**            field        .. Pointer to the integer array
**  exits     EXIT_FAILURE .. memory allocation failed
**
** Bugs:      TODO: File support
**
*/

static int *getfield (const char *namestring, size_t *size)
{
    char *invalid;
```

```
    int *field;

    *size = (size_t) strtoll(namestring,&invalid,10);
    if (*invalid != '\0') { // not a string containing only a number
        FILE *fp = fopen(namestring, "r");
        if(fp==NULL)
            return NULL;
        fclose(fp);
// Confess the truth
fprintf(stderr, "\n\t***sorry, file opened, functionality not"
" yet implemented***\n\n");
return NULL;
    }
    else if (*size == 0) { // zero is not a good array size
        return NULL;
    }
    else { // got an actual number
        field = malloc(*size*sizeof(int));
        if (field==NULL) {
            fprintf(stderr, "%s, line %d: Memory allocation failed."
                            " Aborting...\n", __func__,
                            (int) __LINE__);
            exit(EXIT_FAILURE);
        }
        srand((unsigned) time(NULL));
        unsigned mod = (3*(*size)>*size) ? 3*(*size) : INT_MAX;
        for (size_t i=0; i<*size; i++)
            field[i] = rand()%mod;
    }

    return field;
}
```

***Programmlisting 9.7:*** sorting/insertionsort.h

```
#ifndef INSERTIONSORT
#define INSERTIONSORT
#include <stdlib.h>

int insertionsort (int *array, size_t number);

#endif
```

***Programmlisting 9.8:*** sorting/selectionsort.h

```
#ifndef SELECTIONSORT
#define SELECTIONSORT
#include <stdlib.h>

int selectionsort (int *array, size_t number);

#endif
```

***Programmlisting 9.9:*** sorting/bubblesort.h

```
#ifndef BUBBLESORT
#define BUBBLESORT
#include <stdlib.h>

int bubblesort (int *array, size_t number);

#endif
```

***Programmlisting 9.10:*** sorting/shellsort.h

```
#ifndef SHELLSORT
#define SHELLSORT
#include <stdlib.h>

int shellsort (int *array, size_t number);

#endif
```

***Programmlisting 9.11:*** sorting/quicksort.h

```
#ifndef QUICKSORT
#define QUICKSORT
#include <stdlib.h>

int quicksort_r (int *array, size_t l, size_t r);

inline int quicksort (int *array, size_t number)
{
    return (quicksort_r(array,0,number-1));
}

#endif
```

# A. Einführung in den Umgang mit der Shell

Die im CIP-Pool zur Verfügung stehenden Rechner arbeiten unter dem Betriebssystem *Linux*. Dieses ist eine ursprünglich für PCs entwickelte Variante der Betriebssystem-Familie UNIX, die es schon sehr lange gibt. Ein hervorstechendes Merkmal aus Benutzersicht ist, dass die Interaktion zwischen Nutzer und System primär über die Kommandozeile stattfindet und die grafische Benutzeroberfläche nur ein Aufsatz ist. Das ändert sich zwar allmählich, so dass man die Benutzeroberfläche ähnlich vielseitig einsetzen kann wie unter anderen Betriebssystemen, etwa *MacOS* oder neueren *Windows*-Varianten. Trotzdem kommt man immer wieder an den Punkt, wo man ohne die Kommandozeile nicht weiterkommt.

Außerdem wurde die Programmiersprache ***C*** speziell für UNIX-Betriebssysteme entwickelt, so dass ein Verständnis für die Arbeitsweise der Kommandozeile sinnvoll ist für das Verständnis, warum bestimmte Dinge in ***C*** auf eine bestimmte Weise funktionieren.

Die Kommandozeile läuft unter einer bestimmten *Shell* („*Schale*“ um den *kernel*, den Betriebssystem-*Kern*), in unserem Fall der *bash*. Die Shell stellt dem Benutzer die Möglichkeit zur Verfügung, Programme – wie zum Beispiel den Compiler – zu starten; desweiteren können eigene Befehle und Umgebungsvariablen definiert werden, es kann mit regulären Ausdrücken gearbeitet werden u.v.m.

Im Folgenden betrachten wir einige nützliche Befehle und ein paar Aspekte der Shell.

## A.1. Hilfe

### A.1.1. `man`

Die meisten Programme liefern, wenn sie richtig installiert wurden, eine kurze Dokumentation mit, die so genannte *man-page* (Manual page/Handbuch-Seite), in der beschrieben ist, was das Programm tut und wie man es aufruft. Die man-page erreicht man über die Benutzung des Programms `man`. Um beispielsweise etwas über die Shell *bash* herauszufinden, gibt man auf der Kommandozeile

```
man bash
```

ein. Mit der Space-Taste bzw. Seite nach oben/unten kann man sich durch die Seite bewegen, mit *q* kann man die Seite wieder loswerden. Das Programm `man` benutzt einen sogenannten *Pager* (Seiten-Anzeigeprogramm), um die Bildschirmausgabe zu bewerkstelligen, in der Regel ist das das Programm `less`. Auf der man-page zu `less` erfährt man, wie man besser navigieren oder nach bestimmten Ausdrücken suchen kann usw. Will man herausfinden, welche Möglichkeiten `man` selbst bietet, gibt man natürlich

```
man man
```

ein. Dort erfährt man etwa, dass man mit der Option `-K wort` alle man-pages nach der Zeichenkette `wort` durchsuchen kann und dass es verschiedene *manual sections*, also Handbuch-Abschnitte gibt. Im Abschnitt 1 stehen i.d.R. auf der Shell anwendbare Befehle. Die Funktionen der ***C***-Standard-Bibliothek findet man – wie alle Bibliotheks-Funktionen im Abschnitt 3. Will man etwas über die Funktion `scanf` erfahren, kann man das normal mit

```
man scanf
```

tun, dagegen führt

```
man printf
```

auf die Seite des Befehls `printf` im Abschnitt 1. Gibt man

```
man 3 printf
```

ein, durchsucht `man` nur den Abschnitt 3. Erklärung: `man` zeigt immer nur die erste man-page an, die es findet. Will man alle angezeigt bekommen, muss man die Option `-a` benutzen.

### A.1.2. `info`

In vielen man-pages steht, dass die man-page gar nicht mehr die aktuellste Informationsquelle ist. Stattdessen wird auf eine *texinfo*-Dokumentation hingewiesen, die sich mit dem Programm `info` anzeigen lässt. Hilfe zum Aufruf gibt es mit `man info`, Hilfe zur Benutzung mit `info info`.

### A.1.3. ...und das Netz

Im Internet findet man zu vielen Programmen „richtige“, zum Ausdrucken gedachte Handbücher. Zudem kann man in Newsgruppen oder deren FAQs nach Antworten auf eigene Fragen suchen bzw. auch direkt in einer Newsgruppe eigene Fragen stellen[1].

## A.2. Dateien und Verzeichnisse

Zusammengehörige Informationen, z.B. ein Artikeltext oder ein Bild, werden als *Datei* organisiert. Die Datei enthält dann die Informationen und der Dateiname ermöglicht den Zugriff auf die Informationen. Um Informationen weiter ordnen zu können, gibt es *Verzeichnisse*. Verzeichnisse können Dateien und Verzeichnisse (Unterverzeichnisse) enthalten. Das *root*-Verzeichnis ist die Wurzel des Dateisystems, alle anderen Verzeichnisse liegen unterhalb des root-Verzeichnisses, alle Dateien entweder im root-Verzeichnis oder unterhalb. Ein Verzeichnis kann kein, ein oder mehr Unterverzeichnisse haben, aber es hat (wenn es nicht das root-Verzeichnis ist) genau ein übergeordnetes Verzeichnis, dessen Unterverzeichnis es ist.

[1] Wer mit dem Usenet bzw. Newsgruppen nicht vertraut ist, tut gut daran, erst einmal mit einer Suchmaschine nach *netiquette* bzw. *Netikette* zu suchen und sich über die Spielregeln zu informieren.

Dateien und Verzeichnisse sind eindeutig durch einen *absoluten Pfad* erreichbar. Der absolute Pfad beginnt mit dem root-Verzeichnis `/`, dann folgt die Angabe der Unterverzeichnisse, bis man in dem Verzeichnis angelangt ist, das die gewünschte Datei bzw. das gewünschte Verzeichnis enthält. Zwischen einem übergeordneten und einem untergeordneten Verzeichnis steht als Trennzeichen wiederum der Schrägstrich (*slash*). Beispiel: Das Unterverzeichnis `bin` im Verzeichnis `usr`, das seinerseits im root-Verzeichnis liegt, wird mittels

```
/usr/bin
```

angesprochen, das Programm `bash` mittels

```
/usr/bin/bash
```

angesprochen. Wenn es sehr viele Verzeichnis-Ebenen gibt, ist das Eintippen von absoluten Pfaden recht mühselig. Deshalb gibt es auch die Möglichkeit, mit *relativen Pfaden* zu arbeiten. Diese sind bezogen auf das aktuelle Verzeichnis. Befände man sich im Verzeichnis `/usr`, so könnte man die Datei `bash` auch durch

```
./bin/bash
```

ansprechen. Dabei steht der Punkt `.` für das aktuelle Verzeichnis. Dieser Bezug kann auch weggelassen werden, wenn nicht das aktuelle Verzeichnis selbst gemeint ist, in diesem Fall wäre also auch

```
bin/bash
```

möglich. Das dem aktuellen Verzeichnis übergeordnete Verzeichnis wird mittels `..` angesprochen, mit

```
/usr/bin/..
```

erreicht man also das Verzeichnis `/usr`.

**Hinweis:** Es wird zwischen Groß- und Kleinschreibung unterschieden, d.h. die Datei- bzw. Verzeichnisnamen `datei`, `Datei` und `DATeI` beschreiben verschiedene Objekte! Das ist z.B. unter *MS-DOS* nicht der Fall.

### A.2.1. Eigenschaften/`ls`/`chmod`

Um sich den Inhalt eines Verzeichnisses anzeigen zu lassen, verwendet man das Kommando `ls` (*list*), z.B.

```
ls /usr
```

Anstelle von `ls .` kann man auch nur `ls` schreiben. Dateien und Verzeichnisse, die mit einem Punkt beginnen, werden im Normalfall von `ls` nicht angezeigt. Mit der Option `-a` werden alle Dateien und Verzeichnisse angezeigt. Die Option `-l` führt zur Anzeige zusätzlicher Informationen:

```
mairml@cip20:~> ls -l
total 3831
```

```
-rw-r--r-- mairml  fak         0 Apr 20 11:46 ausg
drwxr-xr-x mairml  fak      2048 Apr 22 14:59 bin
drwxr-xr-x mairml  fak      2048 May 16 17:05 C
drwx------ mairml  fak      2048 Mar 19 15:24 Desktop
drwx------ mairml  fak      2048 May  9 07:42 Mail
-r-xr-xr-x mairml  fak   2058456 Apr 15 16:03 nnew
-r-xr-xr-x mairml  fak   1853216 Apr 15 16:04 nold
drwxr-xr-x mairml  fak      2048 Jul 12 19:36 test
```

Die zweite Spalte zeigt den Besitzer der Datei, in diesem Fall den Benutzer mit dem Konto `mairml`, die dritte zeigt die Gruppe, der die Datei zugeordnet ist: Jeder Benutzer ist in einer oder mehreren Gruppen, Dateien und Verzeichnisse, die ihm gehören, werden auch einer der Gruppen, der er angehoert, zugeordnet. In der vierten Spalte steht die Dateigröße, danach folgt das Datum und die Uhrzeit der letzten Modifikation, die an der Datei vorgenommen wurde, bzw. der Erstellung der Datei oder des Verzeichnisses. Zum Schluss kommt der Datei- oder Verzeichnisname.

In der ersten Spalte stehen *Rechte* und *Typ*. Der Typ ist die erste Stelle; `d` bezeichnet Verzeichniss, `-` Dateien. Danach folgen dreimal drei Stellen mit Rechten. Die ersten drei gelten für den Besitzer, die zweiten drei für die Gruppe und die letzten drei für alle (anderen) Benutzer. Hierbei bezeichnet

- `r` **r**eadable: Die Datei/das Verzeichnis kann gelesen werden, d.h. man kann den Inhalt auslesen.
- `w` **w**ritable: Die Datei/das Verzeichnis kann überschrieben/erweitert werden, d.h. man kann den Inhalt ändern (und auch die Datei löschen).
- `x` e**x**ecutable: Die Datei kann ausgeführt werden, d.h. es handelt sich entweder um ein Programm oder um ein *Skript*. Ein Shell-Skript etwa enthält eine oder mehrere Befehlszeilen, die auch auf der Shell ausführbar sind und in der Datei zu einer Befehlsfolge zusammengefasst sind. Verzeichnisse müssen ausführbar sein, wenn man auf ihren Inhalt zugreifen will.

Die Rechte kann man mit Hilfe des Befehls `chmod` (*change modes*) ändern. Die Kommandozeile lautet

```
chmod Rechte Datei(en)
```

wobei Rechte z.B. in dem Format `u+x` (Besitzer erhält Ausführungsrecht) angegeben werden können. Statt/zusammen mit `u` (Besitzer) kann auch `g` (Gruppe) oder `o` (alle anderen) bzw. `a` (alle) verwendet werden, statt/zusammen mit `x` natürlich auch `r` und `w`; ersetzt man `+` durch `-`, so werden die Rechte weggenommen.

### A.2.2. Verzeichniswechsel (`cd`)/`pwd`

Um relative Pfade verwenden zu können, muss man das Verzeichnis wechseln können; das geschieht mit dem Befehl `cd` (*change directory*):

```
cd PfadZuVerzeichnis
```

macht das Verzeichnis `Pfad` (egal ob relativer oder absoluter Pfad) zum aktuellen Verzeichnis. Um zu erfahren, in welchem Verzeichnis man sich befindet, verwendet man den Befehl `pwd` (*print working directory*).

### A.2.3. Verzeichnisse anlegen (`mkdir`) und löschen (`rmdir`)

Neue Verzeichnisse kann man mittels `mkdir` (*make directory*) anlegen:

```
mkdir PfadZuVerzeichnis
```

Loswerden kann man Verzeichnisse, indem man sie mit `rmdir` (*remove directory*) löscht. `rmdir` funktioniert nur, wenn das Verzeichnis keine Dateien oder Unterverzeichnisse enthält.

### A.2.4. Dateien löschen/rekursives Löschen (`rm`)

Dateien kann man mit dem Befehl `rm` (*remove*) löschen.

```
rmdir PfadZuVerzeichnis
rm Optionen PfadZuDatei1 PfadZuDatei2 ...
rm -r Optionen Pfad1 Pfad2 ...
```

Im Normalfall wird man für jede Datei gefragt, ob man sie wirklich löschen will – das kann sehr lästig sein. Ist man sich absolut sicher, dass man die angegebene(n) Datei(en) löschen will, kann man die Option `-f` verwenden, die die Abfrage deaktiviert. Die Option `-r` löscht die angegebenen Objekte rekursiv, d.h. wenn es sich um ein Verzeichnis handelt, werden erst alle Dateien und Unterverzeichnisse gelöscht und schließlich das Verzeichnis selbst. Die Kombination mit `-f` ist insofern gefährlich, dass man sofort große Mengen ohne weitere Nachfrage löscht. Im Extremfall `rm -rf /` ist auf einen Schlag *alles* weg.

Auch andere Befehle können rekursiv ausgeführt werden (nicht immer heißt die Option dann auch `-r`).

### A.2.5. Dateien/Verzeichnisse erzeugen, kopieren (`cp`) und verschieben (`mv`)

Somit fehlt nur noch eine Möglichkeit, Dateien zu erzeugen. Das geschieht mit Hilfe von Programmen – Textdateien (wie z.B. ***C***-Quellcode-Dateien) werden im Editor geschrieben und dann in ein bestimmtes Verzeichnis gespeichert, Programmdateien werden vom Compiler erzeugt usw.

Manchmal würde man dann gerne eine alte Datei als Ausgangsbasis für eine neue Datei verwenden bzw. eine Sicherungskopie erstellen. Für das Kopieren von Dateien und das rekursive Kopieren von Verzeichnissen gibt es den Befehl `cp` (*copy*):

```
cp Quelldatei(en) Zielpfad
cp -r Quelle(n) Zielpfad
```

Im rekursiven Fall (`cp -r`) kann man sowohl Dateien als auch Verzeichnisse angeben. Soll die Datei/das Verzeichnis unter gleichem Namen kopiert werden, so reicht für `Zielpfad` der Pfad des (existierenden!) Verzeichnisses, in das sie/es kopiert werden soll. Soll der Name sich auch ändern, gibt man den vollständigen Pfad inklusive des neuen Namens an. Nur im ersten Fall ist es möglich, mehrere Quellen anzugeben (im zweiten würden sie ja sonst auf das gleiche Ziel geschrieben).

Manchmal stellt man fest, dass eine Datei oder ein Verzeichnis nicht (mehr) richtig benannt ist. Mit Hilfe von `mv` (*move*) kann man sie verschieben bzw. umbenennen:

```
mv Quell(en) Zielpfad
```

### A.2.6. Home-Verzeichnis/~

Benutzer bekommen unter *Linux* in der Regel ein sogenanntes Home-Verzeichnis, dessen Besitzer sie sind. In diesem Verzeichnis und seinen Unterverzeichnissen haben sie im Allgemeinen alle Rechte bzw. können sie sich selbst mittels `chmod` geben. Das Home-Verzeichnis eines Benutzers `user` lässt sich mittels `~user` ansprechen. Für das eigene Home-Verzeichnis kann man auch nur ~ schreiben.

## A.3. Sich häuslich einrichten/Skripte

Wenn man bestimmte Befehle oft mit bestimmten Optionen aufruft, wäre es ganz praktisch, wenn es dafür eine Abkürzung gäbe. Mit Hilfe des Befehls `alias` lässt sich das bewerkstelligen:

```
alias ll='ls -l'
```

Das würde aber bedeuten, dass man jedes Mal, wenn man eine neue Shell startet, auch wieder alle Abkürzungen von neuem bekanntmachen müsste.

Um das zu vermeiden, gibt es für die *bash* die Möglichkeit, beim Start der Shell ein *Skript* abzuarbeiten; in einem Shell-Skript stehen Shell-Befehle, die der Reihe nach ausgeführt werden. Das Skript, das beim Start der *bash* automatisch abgearbeitet wird, wenn es vorhanden ist, ist `~/.bashrc` und könnte etwa so aussehen:

```
# fuege ~/bin/ dem Suchpfad hinzu
PATH=$PATH:$HOME/bin
export PATH

# wirf doppelte Eintraege aus der History raus
export HISTCONTROL=ignoredups

# Beispiele fuer alias-Anweisungen
alias less='less -r'
alias rm='rm -i'
alias ls='ls -F --color=tty'
alias dir='ls --color=auto --format=vertical'
```

```
alias vdir='ls --color=auto --format=long'
alias ll='ls -l'
alias la='ls -A'
alias l='ls -CF'
```

Mit dem Zeichen # werden Kommentare eingeleitet; der Rest der Zeile wird ignoriert. Mit der Zuweisung `PATH=`PATH:*HOME*`/bin` (keine Leerzeichen um =!) wird der Variablen `PATH` der Wert PATH:*HOME*`/bin` zugewiesen. Mittels `export` wird die Variable global bekanntgemacht (dadurch kann dann auch außerhalb des Skripts und auch von anderen Befehlen darauf zugegriffen werden). Man kann das auch in einem Schritt machen, so wie in der Zeile `export HISTCONTROL=ignoredups`. Mit dem Dollar-Zeichen vor einer Variablen bekommt man ihren Wert. Die Zuweisung von `PATH` hängt also an den alten Wert von `PATH` noch ein `:$HOME/bin` an. Die Variable `HOME` enthält den absoluten Pfad zum Homeverzeichnis `~`. Der Befehl `echo`, der zur Ausgabe einer Zeile mit dem als Argument angegebenen Text führt, kann zur Ausgabe des Wertes einer Variable verwendet werden. Die Aufrufe

```
echo "Wert von PATH:"
echo $PATH
echo $PWD
echo
```

geben eine Zeile mit der Ausgabe `Wert von PATH:`, mit dem Wert von `PATH`, mit dem Wert von `PWD` (i.d.R. das aktuelle Verzeichnis) bzw. eine Leerzeile aus.

In `PATH` ist der System-Suchpfad eingetragen, d.h. dort stehen, durch Doppelpunkte von einander getrennt, die Pfade zu Verzeichnissen, in denen nach Befehlen gesucht wird. Wenn man eigene Befehle oder Skripte erstellt, die man im Verzeichnis `~/bin` ablegt, kann man sie von jedem anderen Verzeichnis aus aufrufen, ohne jedes Mal den gesamten Pfad anzugeben (andernfalls müsste man z.B. auch immer `/bin/ls` an Stelle von nur `ls` schreiben).

**Hinweis:** Unter manchen Betriebssystemen (z.B. *MS-DOS*) ist das aktuelle Verzeichnis im Suchpfad enthalten; das ist unter UNIX aus Sicherheitsgründen normalerweise nicht der Fall. Programme im aktuellen Verzeichnis müssen dann mit führendem `./` aufgerufen werden. Es ist natürlich auch möglich, das aktuelle Verzeichnis in den Suchpfad aufzunehmen, wenn Bequemlichkeit vor Sicherheit geht[2]...

Die History der Shell enthält die letzten ausgeführten Befehle; mit der Pfeil-nach-oben-Taste kann man auf der Kommandozeile jeweils einen Befehl zurückgehen, mit der Pfeil-nach-unten-Taste jeweils vorwärtsgehen. Mit `Strg+R` kann man die History rückwärts durchsuchen, mit `Strg+S` vorwärts.

Eine weitere nützliche Eigenschaft der *bash* ist die *Expandierung* angefangener Eingaben, die sich auf Dateien beziehen: Gibt man beispielsweise `acro` ein und drückt auf die `TAB`-Taste, wird das zu `acroread` (mit anschließendem Leerzeichen) expandiert, wenn der Acro-

[2] Ein weiterer Grund für die Einschränkung ist die Tatsache, dass man bei zu großem Suchpfad Programme gleichen Namens im Pfad haben kann. Dann ist nicht ganz sicher, welches Programm bzw. welche Version des Programms ausgeführt wird. Ein beliebter „Fehler“ ist es, ein Programm namens `test` zu erstellen und sich zu wundern, dass beim Aufruf mittels `test` nichts passiert. Der Befehl `which` zeigt den Pfad zu einer angegebenen ausführbaren Datei an; `which test` gibt uns die Erklärung für das seltsame Verhalten: Es wird `/usr/bin/test` aufgerufen statt `./test`, weil das aktuelle Verzeichnis entweder gar nicht im Suchpfad liegt oder aber *zuletzt* durchsucht wird.

bat Reader installiert ist und als einziges Programm im Suchpfad mit `acro` beginnt. Gäbe es zwei Versionen, die `acroread` und `acroread4` hießen, so würde nur der gemeinsame Teil, `acroread` (ohne anschließendes Leerzeichen) expandiert. Weiß man, welche Varianten vorhanden sind und welche davon man will, kann man jetzt weitertippen. Alternativ kann man aber auch einfach ein zweites Mal TAB eingeben; dann zeigt die *bash* alle Möglichkeiten an. Ähnlich läuft das für Argumente eines Befehls: Mit TAB wird ein angefangener Pfad expandiert, mit einem zweiten TAB gibt es alle Möglichkeiten. Hat man z.B. Dateien `./bilder/bild_0001.gif` bis `./bilder/bild_0013.gif`, so würde die Expansion von `xv ./bilder/b` zu `xv ./bilder/bild_00` führen (wenn es keine mit `b` beginnenden Dateien und Verzeichnisse in `./bilder` gibt), tippt man dann `05` ein, so erfolgt die Expansion auf `xv ./bilder/bild_0005.gif` (plus Leerzeichen). Die Expansion des Pfades bei eindeutigen Verzeichnisnamen erfolgt bis einschließlich des `/`; in unserem Beispiel würde `xv ./b` zu `xv ./bilder/` expandiert.

Die `.bashrc` ist ein besonderes Skript insofern, dass die Befehle ausgeführt werden, als ob sie auf der Kommandozeile eingetippt worden wären. Bei normalen Shell-Skripten ist das nicht der Fall[3].

Ein Skript kann aber auch anders aussehen; in der ersten Übung wird das Skript `build` erstellt, das so aussieht:

```
gcc -o "${1}" "${1}.c"
```

Mit den in der Aufgabe beschriebenen Schritten, macht man das Skript ausführbar und durch Kopieren/Verschieben nach `~/bin` bzw. Anlegen in `~/bin` von überall benutzbar. Ruft man `build` mit der Kommandozeile

```
build name
```

auf, so hat das den gleichen Effekt wie der Aufruf

```
gcc -o name name.c
```

d.h. aus der Quelldatei `name.c` wird von *gcc* ein Programm mit dem Namen `name` erstellt. Kommen noch andere Parameter zum Aufruf von *gcc* hinzu, so wird die Ersparnis noch größer. Mit `$1` wird auf das erste Argument des Skript-Aufrufs zugegriffen; die geschweiften Klammern sorgen im zweiten Fall dafür, dass klar ist, dass `.c` nicht zum Variablennamen gehört, die Anführungszeichen machen klar, dass der Wert von `1` und die Endung zusammengehören. Davon abgesehen könnte man aber auch

```
gcc -o $1 "${1}.c"
```

schreiben. Die Anführungszeichen und geschweiften Klammern sind aber für Skripte sicherer, weil sie auch mit besonderen Zeichen im Variablenwert klarkommen. Der Skriptname steht in der Variable `0`, das *n*-te Argument in Variable `n`. Auf alle Argumente zusammen kann auch mit `$*` zugegriffen werden.

Im Normalfall gibt man auch noch an, von welchem *Interpreter* das Skript abgearbeitet werden soll. Dazu schreibt man in die erste Zeile des Skriptes

[3]Mit Hilfe des Befehls `source` bzw. `.` kann ein Skript aber so ausgeführt werden wie `.bashrc`.

```
#!PfadzumInterpreter
```

Steht nichts dabei, so wird ein Standard-Interpreter aufgerufen (meist `/bin/sh`, eine etwas einfachere Shell als die *bash*); eine „bessere“ Variante des `build`-Skriptes könnte so aussehen:

```
#!/bin/bash
gcc -Wall -std=c99 -pedantic -o $* "${1}.c"
```

Damit könnte man mehr als ein Argument übergeben, z.B.

```
build name -DTEST zweiteDatei.c
```

würde aus den Quelldateien `name.c` und `zweiteDatei.c` das Programm `name` erstellen, wobei auch noch die Compiler-Option `-DTEST` aktiviert wäre.

Während man in der `.bashrc` Befehle und Variablen definiert, sollte man dort keine längeren Programme ausführen, weil die `.bashrc` auch von Programmen ausgeführt wird, die nicht damit zurechtkommen, dass jemand dazwischenfunkt[4]. Will man bestimmte Befehle nach dem Einloggen ausführen, kann man eine Datei namens `.profile` im Home-Verzeichnis anlegen und die Befehle dort hineinschreiben.

Eine weitere nützliche Datei ist `~/.forward`; zu einem Nutzerkonto gehört meistens auch eine E-Mail-Adresse. Dort laufen unter Umständen wichtige Mails auf, die man abzurufen vergisst. Legt man die Datei `~/.forward` an und schreibt dort eine E-Mail-Adresse hinein, so werden Mails an diese Adresse weitergeleitet. Mehrere Einträge (eine E-Mail-Adresse pro Zeile) führen dazu, dass die Mails an alle Adressen weitergeleitet werden. Steht auch die E-Mail-Adresse dabei, an die die Mail ursprünglich ging, so bleibt die Mail vor Ort gespeichert, und wird nur einmal weitergeleitet. Bei mehreren Nutzerkonten sollte man aufpassen, dass man nicht versehentlich mit seinen E-Mails dank `.forward` Pingpong spielt…

## A.4. Reguläre Ausdrücke, Art der Ausführung, Umleitung

### A.4.1. Reguläre Ausdrücke und `for`-Schleifen

Will man sich nicht den gesamten Inhalt eines Verzeichnisses anzeigen lassen, sondern nur eine Auswahl davon, bzw. eine andere Operation auf bestimmte Dateien/Verzeichnisse anwenden, gibt es mit regulären Ausdrücken die Möglichkeit, genau zu beschreiben, welche Dateien/Verzeichnisse man meint. Die regulären Ausdrücke der *bash* unterscheiden sich etwas von denen in Editoren, wodurch sie teilweise etwas einfacher und teilweise etwas schwerer zu handhaben sind. Wir betrachten hier nur zwei Möglichkeiten, Muster zu beschreiben. Will man eine beliebige Anzahl an Zeichen frei lassen, so verwendet man `*`:

```
ls -al *.c
```

[4] Ein Beispiel ist das Programm `scp`, das das sichere Kopieren von Dateien zwischen verschiedenen Rechnern oder Nutzerkonten ermöglicht; es funktioniert nicht, wenn bestimmte zusätzliche Programme in der `.bashrc` ausgeführt werden.

zeigt alle Dateien und Verzeichnisse an, die auf `.c` enden. Will man eine gewisse Auswahl zulassen bzw. ausschließen, verwendet man `[]` bzw. `[^]`:

```
rm datei.[ch]
```

entfernt die Dateien `datei.c` und `datei.h`,

```
rm datei.[^ch]
```

löscht alle Dateien, deren Name mit `datei.` beginnt und mit einem anderen Zeichen als `c` oder `h` weitergeht. Innerhalb der eckigen Klammern kann man auch Regionen angeben durch Verwendung von `-`:

```
chmod u-x *.[0-9chB-F]*
```

nimmt Dateien, die einen Punkt gefolgt von einer Dezimalziffer, einem der Kleinbuchstaben `c` bzw. `h` oder einem der Großbuchstaben `B`, `C`, `D`, `E` oder `F` im Dateinamen haben, das Ausführbarkeitsrecht.

Manchmal wäre es schön, wenn man Befehle, die man nur auf eine Datei anwenden kann, trotzdem für Dateien, die einem bestimmten Muster folgen, verwenden könnte. Mit Hilfe einer `for`-Schleife ist das möglich:

```
for i in *.eps; do gv $i; done
```

zeigt alle `.eps`-Dateien des aktuellen Verzeichnisses mit dem Programm `gv` an. Schleifen über bestimmte Zahlenbereiche sind auch möglich:

```
for ((i=0; $i<6; i=$i+1)); do gv "bild${i}.eps"; done
```

Eine beliebte Anwendung ist auch massenhaftes Kopieren/Umbenennen/Konvertieren etc. von Dateien; dazu muss man Dateiendungen wegstreichen können:

```
for i in *.gif; do convert $i "${i%.gif}.jpg"; done
```

Zwischen `do` und `done` können beliebig viele Befehle und auch weitere Schleifen stehen, siehe auch den nächsten Abschnitt.

### A.4.2. & und ;

Normalerweise kann die Shell nur einen Befehl auf einmal ausführen; man muss warten, bis dieser beendet ist, dann kann man wieder einen neuen Befehl eingeben. Mit einem & am Ende der Kommandozeile kann man dafür sorgen, dass ein Befehl im Hintergrund[5] ausgeführt wird, d.h. die Shell weiterhin zur Verfügung steht, während der Befehl ausgeführt wird. Das ist beispielsweise nützlich, wenn man auf der grafischen Oberfläche arbeitet und einen Editor von der Shell aus starten will.

```
acroread dokumentation.pdf &
```

[5] in einer eigenen Shell

Wenn man das & vergessen hat, gibt es eine andere Möglichkeit, sich „die Shell wiederzuholen“: Man klickt mit der Maus auf die Shell, aus der man das gerade laufende Programm gestartet hat, und gibt Strg+Z ein. Das Programm wird gestoppt und man kann etwas auf der Shell eingeben. Will man anschließend zum Programm zurückkehren, gibt man einfach `fg Programmname` ein (`fg` wie *foreground*/*Vordergrund*) – das Programm wird weiter ausgeführt und die Shell ist wieder weg. Will man die Shell behalten, das Programm aber weiter ausführen, dann gibt man einfach `bg` (*background*/*Hintergrund*) ein – das Programm verhält sich, als hätte man es mit & gestartet.

Das Gegenstück zu & ist das Semikolon; mit `;` zwischen zwei Befehlen sagt man, dass der zweite Befehl auf den ersten warten muss. Zudem kann man damit mehrere Befehle in eine Zeile schreiben. Innerhalb dieser Zeile ist es dann auch möglich, nicht exportierte Variablen von allen Befehlen verwenden zu lassen:

```
i=1; echo; echo "Ich bin die Nummer ${i}"; i=2*$i; echo
```

Eine Schleife in einem Skript, die so aussieht:

```
for i in *.tex; do
    befehl1
    befehl2
    ....
    befehln
done
```

kann man auf der Kommandozeile als

```
for i in *.tex; do befehl1; befehl2; ....; befehln; done
```

schreiben.

### A.4.3. >, 2> und |

Man kann die Ausgabe von Befehlen auch in Dateien *umleiten*:

```
ls -al >ausgabe
```

leitet die Ausgabe, aber nicht die Fehlermeldungen des Befehls `ls -al` in die Datei `ausgabe` um. Danach kann man sich die Ausgabe dann z.B. mit `less ausgabe` anschauen. Manche Programme, unter anderem auch `less` akzeptieren aber auch die Ausgabe eines anderen Programmes als Eingabe. Dann kann man mit Hilfe des *Pipe*-Symbols `|` die Ausgabe weiterleiten:

```
ls -al | less
```

ruft `less` auf, so dass es die Ausgabe von `ls -al` anzeigt.

Die Fehlermeldungen eines Programmes kann man mittels 2> umleiten.

## A.5. Kommandozeilen-Werkzeuge unter UNIX

Es gibt für UNIX-Systeme viele Werkzeuge; hierbei folgen die populärsten Werkzeuge der Philosophie „Tu nur eine Sache, aber tu sie gut“, so dass man oft für einfache Zwecke mehrere Werkzeuge kombinieren muss. Umgekehrt hat das den Vorteil, dass man, weil für jede minimale Anwendung ein mächtiges Werkzeug zur Verfügung steht, fast alles mit dem vorgegebenen Werkzeugsatz realisieren kann. Im Folgenden werden ein paar dieser Werkzeuge aufgeführt.

### A.5.1. Dateien finden – `locate` und `find`

Sucht man eine Datei, deren absoluten Pfad man nicht kennt, aber von der man Teile des Namens kennt, so kann man die Datei versuchen, mit Hilfe von `locate` zu finden; dieses Programm greift auf eine Datenbank zu und liefert alle passenden Einträge zurück. Ist die Datei nicht in der Datenbank enthalten, weil diese sich nicht über alle Verzeichnisse erstreckt (oder nur selten auf den neuesten Stand gebracht wird), dann kann man stattdessen auch das Programm `find` verwenden.

`find` ist ungleich mächtiger als `locate`, da man angeben kann, in welchen Verzeichnissen gesucht werden kann, ob es sich um eine Datei oder ein Verzeichnis handelt, ob die Suche auf Dateien eines bestimmten Alters eingeschränkt werden soll usw. Zudem listet `find` die gefundenen Dateien und Verzeichnisse nicht nur auf, sondern bietet auch die Möglichkeit, mit ihnen etwas bestimmtes zu tun.

Beispiele:

```
find . -name test.c
find . -name "*.c"
find . -type f -name "*test*"
find ~/ -maxdepth 3 -type d -name "Enten*" -exec ls {} \; -print
```

Die erste Zeile sucht im und unterhalb des aktuellen Verzeichnisses nach Dateien oder Verzeichnissen mit Namen `test.c` und gibt alle aus. Im zweiten Fall werden alle Objekte, die auf das Namensmuster `*.c` passen (ohne Anführungszeichen geht es nicht), ausgegeben. Die dritte Suche beschränkt sich auf Dateien, deren Name `test` enthält. Die letzte Suche sucht nur bis zu drei Verzeichnisebenen unterhalb des Home-Verzeichnisses nach Verzeichnissen, die mit `Enten` beginnen und tut mit ihnen, was zwischen `-exec` und `\;` steht. Normalerweise wird dann der Verzeichnisname nicht mehr ausgegeben; wenn wir das aber noch gerne haben wollen, müssen wir separat `-print` angeben. Die ersten drei Suchen könnten wir also auch äquivalent mit der Option `-print` schreiben. Bleibt noch zu klären, was die geschweiften Klammern in der Option `-exec` bewirken: Sie werden durch den Pfad zu einem gefundenen Objekt ersetzt, d.h. die Anweisung gibt den Inhalt aller gefundenen Verzeichnisse aus, wobei nach der Ausgabe des `ls`-Befehls mittels `-print` ausgegeben wird, um welches Verzeichnis es sich dabei gehandelt hat.

### A.5.2. Dinge in Dateien finden (`grep`)

Oft hat man das Problem, dass man den Dateinamen nicht mehr weiß, aber beispielsweise, dass ein bestimmtes, nicht allzu häufiges Wort ebenfalls dort vorkommt. Oder man will wissen, in welchen

Dateien ein bestimmtes Wort oder Wortmuster vorkommt. Hierfür gibt es `grep`.

```
grep authentisch gelaber
```

sucht in der Datei `gelaber` nach der Zeichenkette `authentisch`;

```
grep MeineFunktion *.c
```

sucht in allen Dateien mit der Endung `.c` nach der Zeichenkette `MeineFunktion`;

```
grep "[mM]eine[fF]unktion" *.c
```

sucht nach `MeineFunktion`, wobei `M` und `F` alternativ auch klein geschrieben sein könnten. Mitunter kann es nötig sein, ein gesuchtes Zeichen durch einen *Backslash* (`\`) einzuleiten. Das passiert übrigens auch, wenn man bei einem Skript vergisst, es zum *bash*-Skript zu machen, weil `/bin/sh` oftmals nicht mächtig genug ist.

Häufig sieht man `grep` in Verbindung mit `find`:

```
find Projekt/ -name "*.c" -exec grep "vektor" {} \; -print |less
```

Ist man sich bei Groß- und Kleinschreibung nicht sicher, kann man die Unterscheidung zwischen beiden mit der Option `-i` ausschalten.

```
find Projekt/ -name "*.c" -exec grep -i "vektor" {} \; -print |less
```

findet also auch Zeilen mit `Vektor` oder `VEKTOR`.

### A.5.3. Unterschiede zwischen Dateien finden (`diff`)

Die Unterschiede zwischen zwei Dateien sind ebenfalls relativ oft von Interesse; hierfür verwendet man das Kommando `diff`. Für ***C***-Programme sind die Optionen `-d`, `-u`, `-p` sinnvoll. `diff` gibt die Unterschiede auf die Konsole aus, d.h. wenn man viele Unterschiede erwartet, sollte man die Ausgabe umleiten:

```
diff -dup datei.c.alt datei.c |less
```

Manchmal unterscheidet sich eine Datei von der anderen hauptsächlich durch Einrückungen mit Leerzeichen und Tabulator oder zusätzliche Leerzeilen. Diese Unterschiede kann man mit den Optionen `-b` (lang: `--ignore-space-change`) bzw. `-B` (lang: `--ignore-blank-lines`) unterdrücken, bzw. gleich alle Arten von *white space* (Leerzeichen, Tabulatoren, Zeilenumbrüche) mittels `-w` (lang: `--ignore-all-space`) ignorieren.

## A.6. Abschlussbemerkung

Die besprochenen Befehle und Konzepte sind hier *nicht* vollständig behandelt, es lohnt sich also, in die Dokumentation hineinzuschauen. Viele Befehle bieten übrigens eine kurze Hilfe an, wenn man nur `Befehl -h` oder `Befehl --help` eingibt; dann wird entweder die Manpage oder Teile davon ausgegeben oder zumindest eine Kurzübersicht.

# B. Verknüpfen von *Matlab* und *C*

Die Software-Entwicklung in ***C*** ist manchmal sehr mühsam, wenn man die Fehlerträchtigkeit und Entwicklungsgeschwindigkeit mit der Entwicklung in ***Matlab*** vergleicht. Zudem fehlen in ***C*** Möglichkeiten zur schnellen, unkomplizierten grafischen Ausgabe.

Letzteres ist nicht wirklich ein unlösbares Problem, weil es für jedes Betriebssystem und jede grafische Benutzeroberfläche Bibliotheken gibt, die das erleichtern, aber es wäre schön, wenn man eine Lösung aus einem Guss hätte.

Umgekehrt gibt es viele Skript-Sprachen, die die Möglichkeit bieten, Programme aufzurufen, so dass man sich nur noch auf das Übertragen der Daten zwischen Skript und Programm konzentrieren muss.

***Matlab*** bietet sowohl die Möglichkeit, Programme wie ***Matlab***-eigene Befehle aufzurufen, als auch die Möglichkeit, von ***C*** (***Fortran***, ***Ada***) aus ***Matlab***-Befehle aufzurufen.

Sinnvoll ist das Verwenden von ***C***-Programmen aus ***Matlab*** heraus, wenn die ***Matlab***-Befehle nicht schnell genug sind, z.B. für Echtzeit-Anwendungen.

Umgekehrt macht es sehr viel Sinn, die ***Matlab***-Befehle für grafische Ausgaben zu verwenden, wenn man nur mal eben einen Plot braucht.

Um ein ***C***-Programm zu erzeugen, das von ***Matlab*** aus aufgerufen werden kann, ersetzt man im Wesentlichen `main` durch `mexFunction()` und linkt nicht den Standard-Startup- und Cleanup-Code, sondern ***Matlab***-spezifischen Code.

Diese Aufgabe wird von dem Skript `mex` (von ***Matlab*** aus `mex.m`, unter UNIX-Shells `mex.sh`, unter *Windows* `mex.bat`) erledigt, das das Programm entsprechend kompiliert und linkt.

Das folgende Beispiel zeigt, wie man die Multiplikation mit 2.0 realisieren kann, so dass man von ***Matlab*** aus `bsp1(x)` bzw. `y = bsp1(x)` aufrufen kann. Die Argumente von `mexFunction()` sind der Reihe nach Anzahl der Rückgabeparameter, ein Zeiger auf ein Feld von Rückgabeparametern, Anzahl der Eingabeparameter, Zeiger auf ein Feld von Eingabeparametern.

***Programmlisting B.1:*** matlab/bsp1.c

```
/*************************************************************
**
** bsp1.c
**
** Dieses Beispiel stammt aus dem matlab helpbrowser.
** Wir ersetzen
**  #include <math.h>
**
**  void timestwo (double y[], double x[])
**  {
```

```
**      y[0] = 2.0 * x[0];
**      return;
**  }
**
** durch das mex-Aequivalent
**
*************************************************************/

#include "mex.h"


void timestwo (double y[], double x[])
{
    y[0] = 2.0 * x[0];
}


void mexFunction (int nlhs, mxArray *plhs[],
                  int nrhs, const mxArray *prhs[])
{
    double *x, *y;
    int mrows, ncols;

    if (nrhs != 1) {
        mexErrMsgTxt("Genau ein Eingabeparameter"
                     " erforderlich.");
    } else if (nlhs > 1) {
        mexErrMsgTxt("Zu viele Ausgabevariablen.");
    }

    /* Zeilen- und Spaltenzahl der uebergebenen Matrix */
    mrows = mxGetM(prhs[0]);
    ncols = mxGetN(prhs[0]);

    /* Wir wollen einen Double-Wert, der nicht komplex
    ** ist; ausserdem soll mxn=1x1 sein: */
    if ( !mxIsDouble(prhs[0])
        || mxIsComplex(prhs[0])
         || !(mrows==1 && ncols==1) )
    {
        mexErrMsgTxt("Eingabe muss nichtkomplexer, skalarer"
                     " Double-Wert sein.");
    }

    /* Rueckgabewert entsprechend setzen. */
    plhs[0] = mxCreateDoubleMatrix(mrows, ncols, mxREAL);

    /* Zeiger holen */
    x = mxGetPr(prhs[0]);
    y = mxGetPr(plhs[0]);

    /* Und endlich... */
    timestwo(y,x);
}
```

Das Beispiel stammt im Wesentlichen aus dem ***Matlab***-Helpbrowser (genau wie das Folgende)

und zeigt, wie man die Parameterzahlen und -typen überprüfen kann. Die Rückgabeparameter müssen innerhalb des Programmes alloziert werden. Schließlich muss man sich noch einen Zeiger auf die Werte, die in der Matrix gespeichert sind, besorgen, um mit ihnen arbeiten zu können.

Will man umgekehrt auf ***Matlab***-Funktionen zugreifen, so muss man Eingabeparameter allozieren und diese über den Aufruf von `mexCallMatLab()` an die entsprechende Funktion übergeben. Der Aufruf von `mexCallMatLab()` ist gegenüber `mexFunction()` nur um den Namen der aufzurufenden Funktion erweitert. Im zweiten Beispiel wird eine Matrix mit *x*-Werten gefüllt und eine zweite mittels der ***Matlab***-`sin`-Funktion mit den dazu passenden *y*-Werten. Abschließend wird `plot` aufgerufen. Weil nichts zurückgegeben wird, werden die Matrizen abschließend wieder freigegeben.

***Programmlisting B.2:*** matlab/bsp2.c

```
/*************************************************************
**
** bsp2.c
**
** Dieses Beispiel stammt aus dem matlab helpbrowser.
** Wir ersetzen
**  #include <math.h>
**
**  void timestwo (double y[], double x[])
**  {
**      y[0] = 2.0 * x[0];
**      return;
**  }
**
** durch das mex-Aequivalent
**
*************************************************************/

#include "mex.h"

#define MAX 1000


/* Daten eintragen: Double-Feld, mxArray-Groessen,
** Laenge des Double-Feldes */

void fill (double *ptr, int *ptr_m, int *ptr_n, int max )
{
    /* max gibt die Obergrenze vor, d.h. (*ptr_m)<max */
    *ptr_m = max/2;
    *ptr_n = 1;

    for (int i=0; i < (*ptr_m); i++)
        ptr[i]=i*(4*3.141592654/max);
}

/* gateway function */
void mexFunction( int nlhs, mxArray *plhs[],
                  int nrhs, const mxArray *prhs[] )
{
    int     m, n;
```

```
    mxArray *rhs[2], *lhs[1];

    /* Reellwertige Double-Matrix der Dimension MAXx1 */
    rhs[0] = mxCreateDoubleMatrix(MAX, 1, mxREAL);

    /* Daten mit fill() eintragen */
    fill(mxGetPr(rhs[0]), &m, &n, MAX);
    /* Groesse des Nutzbereichs von rhs[] */
    mxSetM(rhs[0], m);
    mxSetN(rhs[0], n);

    /* <lhs> = sin(<rhs>); je 1 Ein-/Ausgabeparameter */
    mexCallMATLAB(1, lhs, 1, rhs, "sin");

    rhs[1]=lhs[0];
    /* plot(<rhs[0]>,<rhs[1]>); keine Rueckgabe */
    mexCallMATLAB(0, NULL, 2, rhs, "plot");

    /* Speicher freigeben */
    mxDestroyArray(rhs[0]);
    mxDestroyArray(lhs[0]);

    return;
}
```

Wie bei `malloc()` und `free()` sollte man auch bei den dynamischen Speicherallokations-Funktionen von ***Matlab*** aufpassen, dass Speicher nicht zu wenig oder zuoft freigegeben oder auf nicht allozierten Speicher zugegriffen wird.

Wenn man sein Programm vernünftig aufbaut, kann man dafür sorgen, dass sozusagen nur `main` und die Einlese- und Ausgaberoutinen ersetzt werden müssen, wenn das Ganze sowohl von der Kommando-Zeile als auch von ***Matlab*** aus verwendet werden können soll.

Abschließend stellt sich noch die Frage, wie man so ein Programm im schlimmsten Falle debuggen kann – und damit auch, wie man das Kompilieren und Linken beeinflussen kann.

`mex` akzeptiert analog zu *gcc* den Parameter `-g` für den Einbau von Debug-Symbolen. ***Matlab*** muss dann schon mit dem Debugger gestartet werden, z.B. `MatLab -Dgdb`. Außerdem ist es bei manchen ***Matlab***-Releases nicht möglich, die ***Java***-basierte grafische Benutzeroberfläche zu verwenden, d.h. man muss ***Matlab*** dann aus dem Debugger mittels `run -nojvm` starten. In ***Matlab*** selbst schaltet man dann das Debuggen mit `dbmex on` an und ruft die Funktion auf; dann erhält man die Möglichkeit Breakpoints (z.B. bei `mexFunction()`) zu setzen.

Somit verbleibt noch zu klären, wie der Übersetzungsvorgang beeinflussbar ist. `mex` sucht im ***Matlab***-Suchpfad nach der Datei `mexopts.sh`, die die Compiler- und Linkereinstellungen kontrolliert. Hier gibt es einen Abschnitt für jedes Betriebssystem, in unserem Fall ist „glnx86“ von Interesse. Der ***Matlab***-Suchpfad fängt im aktuellen Verzeichnis an und macht dann bei systemweit gültigen Verzeichnissen weiter, d.h. man kann auch seine eigenen Einstellungen verwenden. Außerdem ist es möglich, die Datei mit den Einstellungen namentlich vorzugeben für `mex`, indem man die Option `-f` verwendet.

Abschließend ist der Unterschied zwischen dem systemweiten und dem lokalen, auf ***C*** 99 ausgerichteten, `mexopts.sh` wiedergegeben, wie er sich mit `diff -u6` nach dem Löschen der nicht benötigten Betriebssysteme darstellt:

```
--- mexopts.sh.alt
+++ mexopts.sh
@@ -39,16 +39,16 @@
  glnx86)
 #----------------------------------------------------------------------------
    RPATH="-Wl,--rpath-link,$TMW_ROOT/extern/lib/$Arch,--rpath-link,$TMW_ROOT/bin/$Arch"
 #          gcc -v
 #          gcc version 2.95.2 19991024 (release)
    CC='gcc'
-   CFLAGS='-fPIC -ansi -D_GNU_SOURCE -pthread'
+   CFLAGS='-Wall -std=c99 -pedantic -fPIC -D_GNU_SOURCE -pthread'
    CLIBS="$RPATH $MLIBS -lm"
    COPTIMFLAGS='-O -DNDEBUG'
-   CDEBUGFLAGS='-g'
+   CDEBUGFLAGS='-ggdb3'
 #
 #          g++ -v
 #          gcc version 2.95.2 19991024 (release)
    CXX='g++'
 #          Add -fhandle-exceptions to CXXFLAGS to support exception handling
    CXXFLAGS='-fPIC -ansi -D_GNU_SOURCE -pthread'
@@ -66,13 +66,13 @@
    FOPTIMFLAGS='-O'
    FDEBUGFLAGS='-g'
 #
    LD="$COMPILER"
    LDFLAGS="-pthread -shared -Wl,--version-script,$TMW_ROOT/extern/lib/$Arch/$MAPFILE"
    LDOPTIMFLAGS='-O'
-   LDDEBUGFLAGS='-g'
+   LDDEBUGFLAGS='-ggdb3'
 #
    POSTLINK_CMDS=':'
 #----------------------------------------------------------------------------
    ;;
  esac
 ###############################################################################
```

Über die ***Matlab***-Hilfe lässt sich leicht mehr über den Umgang mit `mex` und den entsprechenden Funktionen (`mx...`) in Erfahrung bringen.

# C. Formatierte Ein- und Ausgabe

Die folgenden Abschnitte sind im Wesentlichen Adaptionen der Aussagen des ISO-Standards 9899:1999, Abschnitt 7.19.6, über `printf` und `scanf` und ihre Verwandten sowie Tabellen und Hinweise auf potenzielle Tücken.

Für die meisten Zwecke genügen die Tabellen auf den Seiten 203 und 204 vollauf – wenn man allerdings spezielle Vorstellungen hat, wie man etwas ausgeben oder einlesen will, ist die formale Beschreibung sinnvoll. Die Formate von `printf` und `scanf` unterscheiden sich, hier sollte man aufpassen.

Die tatsächlich auf dem Zielsystem vorhandenen Versionen der Funktionen sind unter Umständen wesentlich vielseitiger als der Standard vorschreibt. Nichtsdestotrotz ist es sinnvoll, sich auf den Standard zu beschränken. ***C***99-Erweiterungen sind als solche gekennzeichnet.

## C.1. Argumente

Zum besseren Verständnis ist hier noch einmal an die Integer-Promotionen erinnert: `char`, `short` und `int` werden immer zu einem `int` gemacht, wenn sie an eine Funktion übergeben werden, deren Argumente nicht im Voraus bekannt sind[1], `float` und `double` immer zu einem Wert vom Typ `double`. Deswegen spielt es keine Rolle, ob man beim Aufruf von `printf` immer das Ausgabeformat **`%d`** bzw. **`%f`** verwendet, anstatt korrekt **`%hhd`**, **`%hd`** und **`%d`** bzw. **`%f`** und **`%lf`** zu verwenden.

Dagegen spielt es sehr wohl eine Rolle, auf was für einen Typ ein Argument von `scanf` zeigt – die Zeiger sind in der Regel gleich groß, aber die Werte benötigen unterschiedlich viel Speicherplatz. Liest man formal eine `int`-Variable ein an den Platz, wo nur ein `signed char` zur Verfügung steht, schreibt man unter Umständen in andere Variablen hinein, die von der Zuweisung gar nicht berührt werden sollten. Im umgekehrten Fall werden zwar nicht die Speichergrenzen der Variablen verletzt, aber je nachdem, wie Werte im Speicher repräsentiert werden, kann es vorkommen, dass der eingelesene Wert „falsch ankommt“. Das ist meist der Fall, wenn man versehentlich **`%f`** statt **`%lf`** verwendet. Neben falscher Werte kann es auch dazu kommen, dass man eine ungültige Speicherrepräsentation für einen bestimmten Typ erzeugt, wodurch dann nicht nur mit den falschen Werten gearbeitet wird, sondern komplett undefiniertes Verhalten möglich ist.

[1] Das ist sowohl hier der Fall, weil wir eine variable Argumentliste haben, als auch bei der Verwendung von Funktionsprototypen ohne Spezifikation der Argumentliste.

## C.2. Die `printf`-Familie

Für die formatierte Ausgabe gibt es in ***C*** `printf`. Um `printf` verwenden zu können, muss der Header `<stdio.h>` eingeschlossen sein. Kennzeichen von `printf` und verwandten Funktionen ist der Formatstring, in dem angegeben wird, als was die nachfolgenden Parameter zu interpretieren und wie sie auszugeben sind.

```
#include <stdio.h>

int printf    (const char * restrict format, ...);
int fprintf   (FILE * restrict stream,
               const char * restrict format, ...);
int sprintf   (char * restrict s,
               const char * restrict format, ...);
int snprintf  (char * restrict s, size_t n,
               const char * restrict format, ...);

#include <stdarg.h>

int vprintf   (const char * restrict format, va_list arg);
int vfprintf  (FILE * restrict stream,
               const char * restrict format, va_list arg);
int vsprintf  (char * restrict s, const char * restrict format,
               va_list arg);
int vsnprintf (char * restrict s, size_t n,
               const char * restrict format, va_list arg);
```

`printf()` gibt einen String auf der Standardausgabe aus,
`fprintf()` gibt einen String in eine Datei aus[2],
`sprintf()` schreibt das Gewünschte in einen String,
`snprintf()` schreibt das Gewünschte in einen String und begrenzt dabei die Länge der Ausgabe; man sollte *nie* `sprintf()`, sondern immer `snprintf()` benutzen, weil es sonst vorkommen kann, dass man über das Ende des Speichers, der für den zu beschreibenden String reserviert ist, hinausschreibt.

Die mit `v` beginnenden Varianten ermöglichen es, Argumente, die eine eigene Funktion in einer variablen Argumentliste bekommen hat, weiterzureichen. Man könnte die normalen Varianten so bauen, dass sie die variablen Argumentlisten holen und anschließend mit ihnen die entsprechende `v`-Funktion aufrufen. Mehr zum Thema variable Argumentlisten findet sich im Abschnitt 6.6.

#### Allgemeine Form des Formatstrings

Der Formatstring gibt indirekt an, wie viele Argumente anschließend noch übergeben werden (müssen). Hierbei sind nur zu wenige Argumente gefährlich, überzählige führen nicht zu Fehlern.

[2] `printf(....);` ist äquivalent zu `fprintf(stdout, ....);`, siehe Abschnitt 7

Das Format ist eine Folge von Zeichen und besteht aus verschiedenen Anweisungen: Normale Zeichen, die unverändert ausgegeben werden, Zeichen, die mit Hilfe eines vorangestellten Backslashs (\) angegeben werden können, und Konvertierungsanweisungen. Letztere führen dazu, dass die zugehörigen Argumente von hinten geholt werden (diese werden „verbraucht“), konvertiert werden und dann erst ausgegeben werden.

Jede Konvertierungsanweisung wird durch das Prozentzeichen `%` eingeleitet; dem Prozentzeichen folgen der Reihe nach

**Null oder mehr *Flags*** in beliebiger Reihenfolge, die die Bedeutung der Konvertierung ändern.

**minimale Feldbreite (optional)** Hat der konvertierte Wert weniger Zeichen als die Feldbreite, wird von links (bzw. wenn gewünscht von rechts) mit Leerzeichen aufgefüllt. Die Feldbreite kann entweder durch einen Stern (s.u.) oder eine Dezimalzahl angegeben werden. Eine Feldbreite von 0 ist nicht möglich, weil `0` als Flag interpretiert wird.

**Präzision (optional)** Die Präzision gibt für die Ganzzahl-Formate **`d`**, **`i`**, **`o`**, **`u`**, **`x`** und **`X`** die minimale Anzahl von Ziffern, für die Fließkomma-Formate **`a`**, **`A`**, **`e`**, **`E`**, **`f`** und **`F`** die Anzahl an Ziffern nach dem Dezimalpunkt, für die Flieskomma-Formate **`g`** und **`G`** die maximale Anzahl signifikanter Stellen oder für das String-Format **`s`** die maximale Anzahl an Bytes.
Die Präzision wird durch einen Punkt gefolgt von entweder einem Stern (s.u.) oder einer Dezimalzahl angegeben; ist nur ein Punkt angegeben, wird die Präzision als 0 angenommen.

**Längenmodifikator (optional)** beschreibt die Größe des zugehörigen Arguments.

**Format-Spezifikations-Zeichen** gibt die Art der durchzuführenden Konvertierung an.

Feldbreite und/oder Präzision können durch einen Stern angegeben werden. Diesem ist ein Argument vom Typ `int` zugeordnet, das die Stelle angibt. Beispiel:

```
printf("%8.5lf",zahl);
printf("%*.5lf",8,zahl);
printf("%*.*lf",8,5,zahl);
printf("%8.*lf",5,zahl);
```

sind alle gleichwertig. Eine negative Feldlänge wird als `--`Flag plus positive Feldlänge, eine negative Präzision als nicht angegeben gewertet.

### Flags

**`-`** Innerhalb des Feldes wird das Ergebnis der Konvertierung linksbündig angezeigt (sonst rechtsbündig).

**`+`** Das Ergebnis einer vorzeichenbehafteten Konvertierung beginnt immer mit einem Plus- oder Minuszeichen (normalerweise werden nur Minuszeichen angezeigt).

***(Leerzeichen)*** (***C***99) Ist das erste Zeichen einer vorzeichenbehafteteten Konvertierung kein Vorzeichen oder ergibt die Konvertierung einen Teilstring der Länge 0, dann wird dem Ergebnis ein Leerzeichen vorangestellt. Bei Leerzeichen- und `+`-Flag gleichzeitig wird das Leerzeichen-Flag ignoriert.

**#** (*C*99) Das Ergebnis wird in eine alternative Form konvertiert:
Für das Format **o** wird die Präzision erhöht (genau dann, wenn das notwendig ist), um zu erzwingen, dass die erste Ziffer des Ergebnisses eine Null ist; sind Wert und Präzision 0, so wird eine einzelne `0` ausgegeben.
Für das Format **x** (oder **x**) wird ein Ergebnis ungleich 0 mit führendem `0x` versehen.
Für die Formate **a**, **A**, **e**, **E**, **f**, **F**, **g** und **G** enthält das Ergebnis einer Fließkomma-Konvertierung immer einen Dezimalpunkt, selbst wenn keine weiteren Ziffern/Stellen folgen (normalerweise wird in diesem Fall der Dezimalpunkt unterdrückt). Für die Formate **g** und **G** werden zudem angehängte Nullen nicht entfernt.
Für alle anderen Formate ist das Verhalten undefiniert.

**0** (*C*99) Für die Formate **d**, **i**, **o**, **u**, **x**, **X**, **a**, **A**, **e**, **E**, **f**, **F**, **g** und **G** werden führende Nullen zum Auffüllen des Feldes verwendet, anstatt Leerzeichen. Ausnahmen sind bei Fließkomma-Zahlen auftretende Konvertierungen von als unendlich (`inf`) oder undefiniert (`NaN`, *not a number*) interpretierten Werten. Erscheinen die Flags `0` und `-` gleichzeitig, so wird `0` ignoriert. Ist für eines der Formate **d**, **i**, **o**, **u**, **x**, **X** eine Präzision definiert, so wird das `0`-Flag ignoriert.
Für alle anderen Formate ist das Verhalten undefiniert.

#### Längenmodifikatoren

**hh** (*C*99) Ein folgendes Format **d**, **i**, **o**, **u**, **x** oder **X** bezieht sich auf ein Argument vom Typ `signed char` oder `unsigned char` oder ein folgendes Format vom Typ **n** bezieht sich auf einen Zeiger auf ein `signed char`-Argument.

**h** Wie **hh**, aber für `short`s.

**l** Für **d**, **i**, **o**, **u**, **x**, **X** und **n** wie **hh**, aber bezogen auf `long`s.
Für das Format **c**: Argument vom Typ `wint_t`.
Für das Format **s**: Argument vom Typ `wchar_t`.
Für die Formate **a**, **A**, **e**, **E**, **f**, **F**, **g** und **G**: Kein Effekt.

**ll** (*C*99) Wie **hh**, aber für `long long`s.

**j** (*C*99) Wie **hh**, aber für (`u`)`intmax_t`s.

**z** (*C*99) Wie **hh**, aber für `size_t`s (bzw. die dazu passende vorzeichenbehaftete Variante).

**t** (*C*99) Wie **hh**, aber für `ptrdiff_t`s (bzw. die dazu passende vorzeichenlose Variante).

**L** Ein folgendes Format **a**, **A**, **e**, **E**, **f**, **F**, **g** oder **G** bezieht sich auf ein Argument vom Typ `long double`.

Erscheint ein Längenmodifikator mit einem anderen Format als hier angegeben, so ist das Verhalten undefiniert.

**d, i** Das `int`-Argument wird nach dem Stil $[-]dddd$ in eine vorzeichenbehaftete Dezimalzahl umgewandelt. Die Präzision gibt die minimale Anzahl von angezeigten Stellen an; kann der Wert durch weniger Stellen dargestellt werden, so wird mit führenden Nullen aufgefüllt. Normalerweise ist die Präzision 1. Das Ergebnis einer Konvertierung von 0 mit Präzision 0 sind null Zeichen.

**o, u, x, X** Das `unsigned`-Argument wird in vorzeichenlose oktale (**o**), vorzeichenlose dezimale (**u**) oder vorzeichenlose hexadezimale (**x**, **X**) Notation im Stil *dddd* überführt. Für die hexadezimalen Zahlen werden als weitere Ziffern `a`, `b`, `c`, `d`, `e` und `f` (**x**) bzw. `A`, `B`, `C`, `D`, `E` und `F` (**X**) verwendet.
Die Präzision wird wie beim Format **d** bzw. **i** gehandhabt.

**f, F** Ein `double`-Argument wird in eine Dezimal-Fließkommazahl im Stil $[-]ddd.ddd$ umgewandelt, wobei die Nachkommastellen der Präzisionsangabe entsprechen. Ohne Angabe wird von der Präzision 6 ausgegangen. Ist die Präzision 0 und das **#**-Flag nicht angegeben, erscheint kein Dezimalpunkt. Vor einem Dezimalpunkt steht immer mindestens eine Stelle. Der Wert wird auf die entsprechende Stellenzahl gerundet. Ein `double`-Argument, das für eine Unendlichkeit steht, wird in einen der Stile $[-]inf$ oder $[-]infinity$ umgewandelt. Ein `double`-Argument, das für eine nicht definierte Zahl steht, wird in einen der Stile $[-]nan$ oder $[-]nan(Zeichenfolge)$ umgewandelt, wobei *Zeichenfolge* ein compilerabhängiger String ist.
Das Format **F** (***C***99) erzeugt $INF$, $INFINITY$ und $NAN$ anstelle von $inf$, $infinity$ und $nan$.

**e, E** Ein `double`-Argument wird in eine Dezimal-Fließkommazahl im Stil $[-]d.ddd\,e\pm dd$ umgewandelt, wobei es eine Vorkommastelle gibt (die nur für 0 `0` ist) und die Nachkommastellen der Präzisionsangabe entsprechen; $e\pm dd$ bedeutet $\cdot 10^{\pm dd}$. Ohne Angabe wird von der Präzision 6 ausgegangen. Ist die Präzision 0 und das **#**-Flag nicht angegeben, erscheint kein Dezimalpunkt. Vor einem Dezimalpunkt steht immer mindestens eine Stelle.
Der Wert wird auf die entsprechende Stellenzahl gerundet. Das Format **E** erzeugt eine Zahl mit $E$ statt $e$ für den Exponenten.
Der Exponent enthält stets mindestens zwei Stellen und nur so viele mehr, wie zur Darstellung des Exponenten notwendig sind.
Ist der Wert 0, so ist der Exponent Null. Unendlichkeiten und `NaN`s werden wie beim Format **f** bzw. **F** behandelt.

**g, G** Ein `double`-Argument wird in eine Dezimal-Fließkommazahl im Stil **f** oder **e** (bzw. **F** oder **E**) umgewandelt, wobei die Präzision hier die Anzahl der *gültigen* Stellen beschreibt. Eine Präzision von 0 wird wie 1 behandelt.
Der verwendete Stil hängt vom konvertierten Wert ab. Stil **e** (bzw. **E**) wird nur dann verwendet, wenn der Exponent der resultierenden Dezimalzahl kleiner als $-4$ oder größer oder gleich der angegebenen Präzision ist. Angehängte Nullen im Nachkommabereich werden entfernt (wenn nicht das **#**-Flag gesetzt ist). Ein Dezimalpunkt erscheint nur, wenn er von einer Ziffer gefolgt wird.
Unendlichkeiten und `NaN`s werden wie beim Format **f** bzw. **F** behandelt.

**a, A** (*C*99) Wie **e** (bzw. **E**) für Hexadezimal-Fließkommazahlen im Stil $[-]0xh.hhh\ p \pm d$. Es werden die gleichen zusätzlichen Ziffern wie für **x** (bzw. **X**) verwendet, bei **X** werden `X` und `P` ausgegeben.

**c** Ist kein **l**-Längenmodifikator angegeben, wird das `int`-Argument in ein `unsigned char` konvertiert und das resultierende Zeichen wird ausgegeben. Mit **l** wird das `wint_t`-Argument konvertiert wie ein Zeichen einer **ls**-Formatangabe.

**s** Ist kein **l**-Längenmodifikator angegeben, so ist das Argument ein Zeiger auf das erste Element eines Strings. Dieser String wird ausgegeben (bis auf den Terminator `'\0'`).
Ist die Präzision angegeben, dann werden höchstens so viele Bytes ausgegeben. Ist die Präzision garantiert kleiner als die Stringlänge, so kann auch ein statt des Strings nur ein `char`-Feld (ohne Stringterminator) übergeben werden.
Mit **l** zeigt das Argument auf das Anfangselement eines Feldes vom Typ `wchar_t`. Die `wchar_t`-Zeichen werden in Multibyte-Zeichen umgewandelt bis hin zum Stringterminator. Die Präzision begrenzt wiederum die Anzahl der auszugebenden Bytes.

**p** Das Argument wird als `void *` aufgefasst. Der Wert des Zeigers wird auf eine durch die Implementierung definierte Weise dargestellt, z.B. als hexadezimale Speicheradresse.

**n** Das Argument ist ein Zeiger auf ein `signed`-Objekt, in das die Anzahl an Zeichen hineingeschrieben wird, die bis jetzt geschrieben wurde. Kein Argument wird konvertiert, aber eines wird verbraucht. Flags, Feldbreiten- und Präzisionsangaben führen zu undefiniertem Verhalten.

**%** Ein Prozentzeichen wird ausgegeben. Kein Argument wird konvertiert. Die komplette Formatangabe ist **%%**.

**Wichtig:** Die Feldbreite ist *keine* Obergrenze für die Anzahl ausgegebener Zeichen! Deshalb ist `snprintf` immer `sprintf` vorzuziehen.

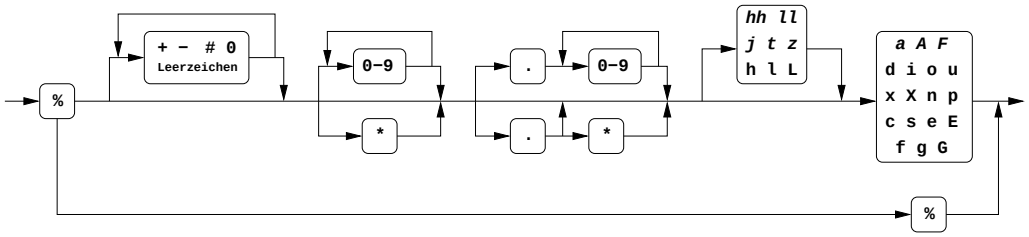
Die Fließkomma-Formate führen nur für eine Präzision innerhalb der gültigen Stellenzahl garantiert zu korrekt gerundeten Zahlen.

#### Rückgabewert

Funktionen der `printf`-Familie geben die Anzahl ausgegebener Zeichen zurück, oder aber einen negativen Wert, wenn ein Ausgabe oder Zeichenkodierungsfehler aufgetreten ist.

## C.3. Die `scanf`-Familie

Das formatierte Einlesen übernimmt in *C* `scanf`, das wie `printf` den Header `<stdio.h>` benötigt. Der Formatstring ist ähnlich aufgebaut wie bei `printf`, allerdings sind die nachfolgenden Argumente nicht „normale“ Variablen, sondern die zugehörigen Adressen, damit `scanf` „weiß“, wohin die gelesenen Informationen gespeichert werden können, siehe auch den Abschnitt über Zeiger, 4.7.

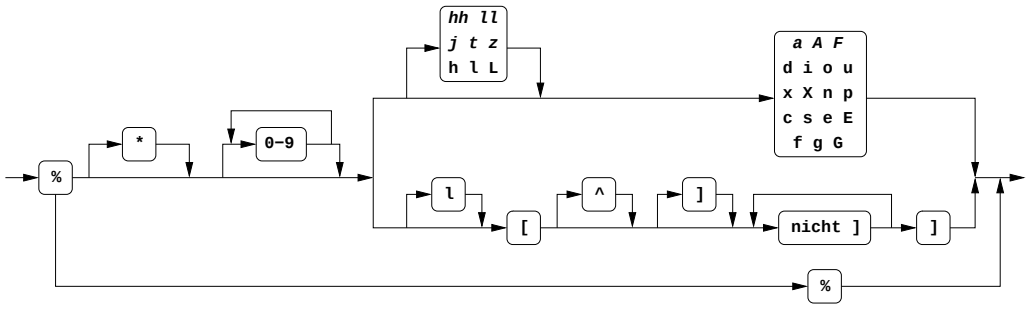



| **Aufbau:** | *%(Flags)(Feldbreite)(.Präzision)(Länge)Format* |
|---|---|

| **Format (+Länge)** | **Argument vom Typ** |
|---|---|
| **%d, %i** | `int` |
| **%ld** | `long int` |
| **%lld** | `long long int` |
| **%hd** | `short int` |
| **%hhd** | `signed char` |
| **%u** | `unsigned int` |
| **%lu** | `unsigned long int` |
| **%ju** | `uintmax_t` |
| **%zu** | `size_t` |
| **%o** | `unsigned int` (oktal ausgegeben) |
| **%x, %X** | `unsigned int` (hexadezimal ausgegeben) |
| **%f, %lf, %F, %lF** | `float`, `double` (Fließkommadarstellung) |
| **%Lf** | `long double` |
| **%e, %le, %E, %lE** | `float`, `double` (Exponentialdarstellung) |
| **%g, %lg, %G, %lG** | `float`, `double` (kürzeste Darstellung) |
| **%a, %A** | `float`, `double` (Hexadezimal-Exponential-Darstellung) |
| **%c** | `char` (einzelnes Zeichen) |
| **%s** | `char *` (String) |
| **%p** | Zeiger (`void *`) |
| **%n** | Zeiger auf `int`, keine Ausgabe, sondern Abspeichern der bis jetzt ausgegebenen Zeichen. |
| **%%** | (kein Typ): %, Prozentzeichen wird ausgegeben. |

| **Flags** | **Wirkung** |
|---|---|
| **-** | linksbündige Ausrichtung im Feld |
| **+** | Positives Vorzeichen wird explizit als + ausgegeben |
| *(Leerzeichen)* | Bei positivem Vorzeichen wird Leerzeichen ausgegeben |
| **#** | Typ der Ausgabe klar kennzeichnen (oktale oder hexadezimale Ganzzahl bzw. Fließkommazahl) |
| **0** | Nullen statt Leerzeichen zum Auffüllen der Feldbreite |

**Tabelle C.1.:** Das wichtigste über `printf`-Formate. Oben: Aufbau des Formats (schräggedruckt: ***C***99-Neuerungen), unten: Erklärungen.




| **Aufbau:** | *%(*)(Feldbreite)(Länge)Format* |
|---|---|

| **Format (+Länge)** | **Argument vom Typ Zeiger auf** |
|---|---|
| **%*....** | kein zugeordnetes Argument; Format-Anweisung wird abgearbeitet, Zuweisung wird unterdrückt |
| **%d** | `int` |
| **%ld** | `long int` |
| **%lld** | `long long int` |
| **%hd** | `short int` |
| **%hhd** | `signed char` |
| **%u** | `unsigned int` |
| **%lu** | `unsigned long int` |
| **%ju** | `uintmax_t` |
| **%zu** | `size_t` |
| **%o** | `unsigned int` (als Oktalzahl einlesen) |
| **%x, %X** | `unsigned int` (als Hexadezimalzahl einlesen) |
| **%i** | `int` (je nach Darstellung als Oktal-, Dezimal- oder Hexadezimalzahl einlesen) |
| **%a, %A, %e, %E, %f, %F, %g, %G** | `float` |
| **%lf** etc. | `double` |
| **%Lf** etc. | `long double` |
| **%c** | `char` (Array der angegebenen Länge, i.d.R 1) |
| **%s** | `char` (String) |
| **%[....]** | `char` (String aus Zeichen des Scansets, angegeben in der Liste) |
| **%[^....]** | `char` (String aus Zeichen des Scansets, Komplement durch Liste gegeben) |
| **%[]....]** | `char`; Scanset enthält `]` |
| **%[^]....]** | `char`; Scanset enthält nicht `]` |
| **%p** | `void` (Adressdarstellung) |
| **%n** | `int`, kein Einlesen von Zeichen, sondern Abspeichern der bis jetzt gelesenen Zeichen. |
| **%%** | (kein Typ): %, Prozentzeichen wird ausgegeben. |

**Tabelle C.2.:** Das wichtigste über `scanf`-Formate. Oben: Aufbau des Formats (schräggedruckt: ***C***99-Neuerungen), unten: Erklärungen.

```
#include <stdio.h>

int scanf   (const char * restrict format, ...);
int fscanf  (FILE * restrict stream,
             const char * restrict format, ...);
int sscanf  (const char * restrict s,
             const char * restrict format, ...);

#include <stdarg.h>

int vscanf  (const char * restrict format, va_list ap);
int vsscanf (const char * restrict s,
             const char * restrict format, va_list ap);
int vfscanf (FILE * restrict stream,
             const char * restrict format, va_list ap);
```

`scanf()` liest von der Standardeingabe,
`fscanf()` liest aus einer Datei[3],
`sscanf()` liest aus einem String.
Wegen der unten besprochenen Eigenheiten von `scanf()`/`fscanf()` ist oft das Einlesen einer Zeile (mit `fgets()`) in einen String sinnvoll, der dann mit `sscanf()` gescannt wird.

Für die mit `v` beginnenden Varianten gilt das gleiche wie bei `printf` und Co.

### C.3.1. Anweisungen und allgemeine Form des Formatstrings

Der Formatstring gibt indirekt an, wie viele Argumente anschließend noch übergeben werden (müssen). Hierbei sind nur zu wenige Argumente gefährlich, überzählige führen nicht zu Fehlern.

Das Format ist eine Folge von Zeichen und besteht aus verschiedenen Anweisungen: Ein oder mehr *white space*-Zeichen[4], normale Zeichen und Konvertierungsanweisungen. Letztere führen dazu, dass die zugehörigen Argumente von hinten geholt werden (diese werden „verbraucht“), dann wird die Konvertierung von Zeichen vorgenommen und in die Argumente (die ja Adressen sind) geschrieben.

Jede Konvertierungsanweisung wird durch das Prozentzeichen % eingeleitet; dem Prozentzeichen folgen der Reihe nach

*** (optional)** Unterdrücken einer Zuweisung (es wird gescannt, aber nirgendwohin gespeichert).

**maximale Feldbreite (optional)** Dezimalzahl ungleich Null; für Zeichen.

**Längenmodifikator (optional)** beschreibt die Größe des Objekts, auf das das entsprechende Argument zeigt.

[3] `scanf(....);` ist äquivalent zu `fscanf(stdin, ....);`, siehe Abschnitt 7
[4] white spaces im ***C***-Sinne sind alle Zeichen, die von der Funktion `isspace()` als solche erkannt werden, in der Regel sind das Leerzeichen, Tabulator und Zeilenumbruch.

**Format-Spezifikations-Zeichen** gibt die Art der durchzuführenden Konvertierung an.

`scanf()` und Co. führen alle Anweisungen der Reihe nach aus. Bei einem Fehler erfolgt der vorzeitige Abbruch, was sich im Rückgabewert wiederspiegelt. Fehler fallen entweder in die Kategorie Eingabefehler (Kodierungsfehler oder keine weiteren Zeichen aus dem Eingabestrom/-string) oder Übereinstimmungsfehler (die Zeichen geben das richtige Format nicht her).

Eine Anweisung, die aus einem (oder mehreren) white spaces besteht, wird ausgeführt, indem bis zum ersten Nicht-white space-Zeichen gelesen wird bzw. bis keine weiteren Zeichen mehr gelesen werden können.

Eine Anweisung, die aus einem (Multibyte-)Zeichen besteht, wird ausgeführt, indem das nächste Zeichen des Eingabestroms/-strings gelesen wird. Weicht dieses Zeichen von dem angegebenen ab, so ist ein Fehler aufgetreten. Anders gesagt: `scanf` versucht genau das Zeichen aus der Anweisung einzulesen.

Eine Anweisung, die sich auf ein Format/eine Konvertierungsanweisung bezieht, definiert eine Menge passender Eingabesequenzen (wie unten beschrieben); sie wird in den folgenden Schritten ausgeführt:

- White space-Zeichen werden übersprungen, es sei denn, das Format enthält einen der Spezifikatoren **`[`**, **`c`** oder **`n`**.
- Ein Eingabeobjekt wird vom Strom/aus dem String gelesen, es sei denn, das Format ist **`n`**. Ein Eingabeobjekt ist definiert als die längstmögliche Folge von Zeichen, die nicht eine angegebene Feldweite überschreitet und die eine zum Format passende Eingabesequenz ist. Das erste Zeichen nach dem Eingabeobjekt bleibt ungelesen (falls vorhanden). Ist die Länge des Eingabeobjekts 0, ist ein Fehler aufgetreten; sofern das nicht am Eingabestrom/-string liegt, handelt es sich dann um einen Übereinstimmungsfehler.
- Außer im Falle des Formates **`%`** (oder im Falle des Formates **`n`**, Anzahl der eingelesenen Zeichen) wird das Eingabeobjekt dann gemäß der Konvertierungsanweisung konvertiert. Falls das Eingabeobjekt eine unpassende Zeichenfolge ist, scheitert die Ausführung der Anweisung (Übereinstimmungsfehler). Wurde die Zuweisung nicht mittels **`*`** unterdrückt, so wird das Ergebnis der Konvertierung an die Adresse geschrieben, die durch das nächste noch unverbrauchte Argument gegeben ist. *Wichtig:* Ist das Objekt, auf das das Argument zeigt, vom falschen Typ bzw. kann das Ergebnis der Konvertierung von diesem Objekt nicht dargestellt werden, ergibt sich ein undefiniertes (und damit nach MURPHYS Gesetz im ungünstigsten Moment falsches) Verhalten.

#### Längenmodifikatoren

**`hh`** (*C*99) Ein folgendes Format **`d`**, **`i`**, **`o`**, **`u`**, **`x`**, **`X`** oder **`n`** bezieht sich auf ein Argument vom Typ Zeiger auf `signed char` oder `unsigned char`.

**`h`** Wie **`hh`**, aber für `short`s.

**`l`** Für **`d`**, **`i`**, **`o`**, **`u`**, **`x`**, **`X`** und **`n`** wie **`hh`**, aber bezogen auf `long`s.
Für die Formate **`a`**, **`A`**, **`e`**, **`E`**, **`f`**, **`F`**, **`g`** und **`G`**: Argument vom Typ Zeiger auf `double` (statt auf

`float`).
Für die Formate **`c`**, **`s`** und **`[`**: Argument vom Typ Zeiger auf `wchar_t`.

**`ll`** (***C***99) Wie **`hh`**, aber für `long long`s.

**`j`** (***C***99) Wie **`hh`**, aber für (`u`)`intmax_t`s.

**`z`** (***C***99) Wie **`hh`**, aber für `size_t`s (bzw. die dazu passende vorzeichenbehaftete Variante).

**`t`** (***C***99) Wie **`hh`**, aber für `ptrdiff_t`s (bzw. die dazu passende vorzeichenlose Variante).

**`L`** Ein folgendes Format **`a`**, **`A`**, **`e`**, **`E`**, **`f`**, **`F`**, **`g`** oder **`G`** bezieht sich auf ein Argument vom Typ Zeiger auf `long double`.

Erscheint ein Längenmodifikator mit einem anderen Format als hier angegeben, so ist das Verhalten undefiniert.

#### Formate

**`d`** (Optional) vorzeichenbehaftete Dezimalzahl; ohne Längenmodifikator ist das zugehörige Argument ein Zeiger auf `int`. Die gescannten Zeichen werden behandelt wie ein Aufruf von `strtol()` mit dem Basis-Argument 10.

**`i`** (Optional) vorzeichenbehaftete Dezimalzahl; ohne Längenmodifikator ist das zugehörige Argument ein Zeiger auf `int`. Die gescannten Zeichen werden behandelt wie ein Aufruf von `strtol()` mit dem Basis-Argument 0, d.h. Zahlen, die mit `0x` eingeleitet werden, werden als Hexadezimalzahlen, Zahlen, die mit `0` (ohne folgendes `x`) eingeleitet werden, als Oktalzahlen und alle anderen Zahlen als Dezimalzahlen behandelt.

**`o`, `u`, `x`, `X`** Das Argument ist vom Typ Zeiger auf `unsigned` und der zurückgeschriebene Wert stammt entweder aus einer Konversion aus dem oktalen (**`o`**), dezimalen (**`u`**) oder hexadezimalen (**`x`**, **`X`**) Zahlensystem.

**`f`, `F`, `e`, `E`, `g`, `G`, `a`, `A`** (Optional) vorzeichenbehaftete Fließkommazahl; ohne Längenmodifikator ist das zugehörige Argument ein Zeiger auf `float`. Die gescannten Zeichen werden behandelt wie bei einem Aufruf von `strtod()`. (Die Formate **`a`**, **`A`**, **`F`** gibt es mit ***C***99.)

**`c`** Zeichenfolge von genau Feldbreite Zeichen (normalerweise 1). Ist kein **`l`**-Längenmodifikator angegeben, so wird das Argument als Zeiger auf ein `char`-Feld aufgefasst, das groß genug für die gelesenen Zeichenfolge ist (ohne Stringterminator). Mit **`l`** wird von einem `wchar_t`-Feld und einer entsprechenden Interpretation der gelesenen Bytes ausgegangen.

**`s`** Folge von Nicht-white space-Zeichen. Ist kein **`l`**-Längenmodifikator angegeben, so ist das Argument ein Zeiger auf den Anfang eines `char`-Feldes, das groß genug ist für die gelesenen Zeichen plus den Stringterminator, der automatisch angehängt wird. Mit **`l`** zeigt das Argument auf den Anfang eines Feldes vom Typ `wchar_t`, die gelesenen Bytes werden wiederum entsprechend interpretiert.

**[** Nichtleere Folge von Zeichen aus einer Menge erwarteter Zeichen, dem *Scanset*. Ist kein **l**-Längenmodifikator angegeben, so ist das Argument ein Zeiger auf den Anfang eines `char`-Feldes, das groß genug ist für die gelesenen Zeichen plus den Stringterminator, der automatisch angehängt wird. Mit **l** zeigt das Argument auf den Anfang eines Feldes vom Typ `wchar_t`, die gelesenen Bytes werden wiederum entsprechend interpretiert.
Die Formatspezifikation umfasst `[`, alle folgenden Zeichen und die abschließende Klammer `]`. Die Zeichen zwischen den Klammern bilden das *Scanset*, es sei denn, das Zeichen nach der linken Klammer ist ein Zirkumflex (`^`): In diesem Fall enthält das Scanset alle Zeichen, die *nicht* zwischen Zirkumflex und rechter Klammer erscheinen.
Beginnt das Format mit `[]` oder `[^]`, so ist das `]`-Zeichen in der Liste enthalten und die schließende rechte Klammer kommt erst später. Ist ein `-` zwischen den Klammern und folgt es nicht der öffnenden Klammer bzw. dem Zirkumflex oder geht unmittelbar der schließenden Klammer voraus, dann wird es auf besondere, implementierungsabhängige Weise interpretiert. In der Regel wird damit eine Menge von Zeichen zwischen dem Zeichen links und rechts bezeichnet, d.h. `[0-8a-zA-F]` scannt die Ziffern von 0 bis 8, alle Kleinbuchstaben des Alphabets und die Großbuchstaben von A bis F.

**p** Das Argument wird als `void *` aufgefasst. Es wird eine Adresse in dem Format eingelesen, in dem sie durch die `printf`-Familie ausgegeben würde.

**n** Das Argument ist ein Zeiger auf ein `signed`-Objekt, in das die Anzahl an Zeichen hineingeschrieben wird, die bis jetzt gelesen wurde. Kein weiteres Zeichen wird durch dieses Format eingelesen. Dementsprechend ist diesem Format kein Eingabeobjekt zugeordnet (es wird also auch bei Erfolg nicht gezählt). Zuweisungsunterdrückung führt zu undefiniertem Verhalten.

**%** Ein Prozentzeichen wird einzulesen versucht. Das komplette Format ist **%%**.

**Wichtig:** White space am Ende der Eingabe, insbesondere auch Zeilenumbrüche, werden nicht gelesen, wenn nicht per Anweisung gefordert. Das führt bei der Verwendung von `scanf` oft zu scheinbar nicht nachvollziehbaren Ergebnissen. Oft leert man deshalb sicherheitshalber den Eingabepuffer nach einem Aufruf von `scanf`:

```
....
scanf("%lf%s", &mydouble, mystring);

while (getchar() != '\n') {}
printf("Naechste Eingabe: "); fflush(stdout);
scanf("%s", mystring2);
....
```

Weil es möglich ist, dass eine Datei in die Standardeingabe umgeleitet wird, verwendet man in der Regel eine Form, die auch Lesefehler oder ein „Dateiende“ besser akzeptiert:

```
....
int tmp;
while ( (tmp = getchar()) != EOF )
    if (tmp == '\n')
        break;
```

```
....
```

### Rückgabewert

Tritt ein Eingabefehler auf, *bevor* ein Eingabeobjekt eingelesen und die entsprechende Konvertierung und Zuweisung durchgeführt wurde, wird der Wert der symbolischen Konstante `EOF` zurückgegeben. Ansonsten wird die Anzahl der erfolgreich bearbeiteten Eingabeobjekte/Formate zurückgegeben, die niedriger sein kann als die gewünschte Zahl (oder sogar Null).

### Beispiele

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

int i;
float x;
char name[50];
fscanf(stdin, "%2d%f%*d %[0123456789]", &i, &x, name);
```

mit der Eingabe `56789 0123 56a72` weist `i` den Wert 56 und `x` den Wert 789.0 zu, überspringt `0123` und weist `name` die Zeichenfolge `56\0` zu. Das nächste Zeichen, das vom Strom gelesen wird, ist `a`.

Es ist sinnvoll, zu überprüfen, ob man überhaupt die entsprechende Anzahl an Eingabeobjekten bekommen hat, bevor man mit den entsprechenden Variablen weiterarbeitet:

```
#include <stdio.h>
....
int d1, d2, n1, n2, i;
i = sscanf("123", "%d%n%n%d", &d1, &n1, &n2, &d2);
```

Hier wird `d1` der Wert 123 zugewiesen, dann `n1` die Anzahl gelesener Zeichen, 3, desgleichen `n2` und dann wird wegen des Eingabeendes der Variable `d2` kein neuer Wert zugewiesen; die Anzahl bearbeiteter Eingabeobjekt wird `i` zugewiesen und ist 1 (**`%n`** führt nicht zu einem Lesen aus dem Eingabestrom/-string, weswegen der Rückgabewert sich dadurch nicht ändert, s.o.).

### POP's device

***C*** bietet keine vernünftige Funktion, um eine Zeile in einen String einzulesen, so dass es keine Probleme gibt, wenn die Zeile zu lang ist. Das am wenigsten fehlerträchtige Verhalten ist das Einlesen bis zur Höchstzahl der Zeichen und Wegwerfen des potenziellen Restes bis einschließlich des Zeilenumbruchs[5]. Eine Funktion, die das tut, kann man sich einfach mit Hilfe von `getchar()` basteln. Eine andere Möglichkeit ist POP*'s device*[6]:

```
#define NAMELEN       30    /* NAMELEN ist beliebig */
```

[5] Alternativ kann man natürlich auch immer mit alloziertem Speicher arbeiten und ggf. den Speicher vergrößern.
[6] Benannt nach Dan Pop, einem regular von `comp.lang.c`, der zeigen wollte, dass `scanf` sicherer ist als `fgets`.

```
#define STRINGIZE(s)  # s
#define XSTR(s)       XSTR(s)
....

    int rc;
    char string[NAMELEN + 1];

    /* Bis zu NAMELEN Nicht-'\n'-Zeichen in String lesen,
    ** etwaige sonstige Nicht-'\n'-Zeichen lesen. */
    rc = scanf("%" XSTR(NAMELEN) "[^\n]%*[^\n]", string);
    /* Wenn wir nicht am Dateiende sind, ist das naechste
    ** Zeichen '\n'. */
    if (!feof(stdin)) {
        getchar();  // '\n' aus Eingabestrom entfernen
    }
    /* Haben wir auch tatsaechlich was gelesen? */
    if (rc != 1) {
        /* Fehlerbehandlung */
    }
    ....
```

# Literaturverzeichnis


[DD03] H.M. Deitel and P.J. Deitel. *C: How to Program*. Prentice Hall, 2003.

[GBD03] J. Goll, U. Bröckl, and M. Dausmann. *C als erste Programmiersprache. Vom Einsteiger zum Profi*. Teubner Verlag, 2003.

[GGW98] J. Goll, U. Grüner, and H. Wiese. *C als erste Programmiersprache*. Teubner Verlag, 1998.

[Her02] H. Herold. *C Kompaktreferenz*. Addison-Wesley, 2002.

[KR90] B.W. Kernighan and D.M. Ritchie. *Programmieren in C*. Hanser Fachbuch, 1990.

[Krü98] G. Krüger. *Go To C-Programmierung*. Addison-Wesley, 1998.

[Sed92] R. Sedgewick. *Algorithmen in C*. Addison-Wesley, 1992.

[Sed97] R. Sedgewick. *Algorithms in C, Parts 1-4: Fundamentals, Data Structures, Sorting, Searching*. Addison-Wesley Professional, 1997.

[Sed01] R. Sedgewick. *Algorithms in C, Part 5: Graph Algorithms*. Addison-Wesley Professional, 2001.

[Sum] Web: S. Summit. *Materialien und Links*. http://www.eskimo.com/~scs/.

[Weba] Web. *comp.lang.c FAQ*. http://www.eskimo.com/~scs/C-faq/top.html, Maintainer: S. Summit.

[Webb] Web. *de.comp.lang.c FAQ*. http://home.pages.de/~c-faq/, Maintainer: J. Schoof.


**Anmerkung:** [GBD03] ist eine überarbeitete Fassung von [GGW98], letzteres aber in der UB vorhanden.