
Programmieren mit C und Python

Release 1.0

Jens Pönisch

25.12.2020

Inhalt:

1	Einführung	1
2	Grundlagen von C	9
3	Einfache Funktionen	23
4	Logische Ausdrücke	35
5	Schleifen	45
6	Fließpunktzahlen	53
7	Felder	65
8	Structs	85
9	Zeichen und Strings	97
10	Sortieren	111
11	Graphen	125
12	Einführung in Python	141
13	Python-Listen	159
14	Python-Strings	171
15	Python-Klassen	179
16	Weitere Aufgaben	193
17	Begriffe und Abkürzungen	201
	Literaturverzeichnis	203
	Stichwortverzeichnis	205

Inhalt

- *Kursinhalt*
- *Syntaxdarstellung*
- *Compiler und Interpreter*
- *Geschichte*
- *Arbeitsumgebung*
- *Wichtige Befehle der Linux-Kommandozeile*
- *Werkzeuge*
- *Testen der Umgebung*
- *Aufgaben*

1.1 Kursinhalt

Parallel zur Veranstaltung *Algorithmen und Programmierung* beschäftigen wir uns mit der Umsetzung von Algorithmen in den Programmiersprachen *C* und *Python*.

Die Grundlagen der Umsetzung betrachten wir anhand der maschinennahen Sprache *C*, wobei wir uns auf die grundlegenden Konzepte beschränken und auf den Einsatz von Bibliotheken weitgehend verzichten.

Moderne Konzepte wie die Grundlagen der Objektorientierten Programmierung (*OOP*) betrachten wir an der Sprache *Python*, die uns eine deutlich höher Abstraktion zwischen Hardware und algorithmischer Umsetzung bietet. Diese Konzepte begegnen uns später wieder in der Programmiersprache *Java*.

1.2 Syntaxdarstellung

Im Script wird auf eine vereinfachte Syntaxdarstellung verwendet. Wir benutzen folgende Symbole:

keyword Der Text ist exakt so wie angegeben zu übernehmen.

variable Der variable Teil ist passend zu ersetzen.

[optional] Dieser Teil ist optional und kann entfallen.

... Der vorige Teil kann analog wiederholt werden.

Kommentar Kommentar zur Erläuterung, darf nicht übernommen werden.

Für komplette Codebeispiele wurde die eingebaute Syntaxhervorhebung des Dokumentationstools *Sphinx* verwendet.

1.3 Compiler und Interpreter

Um einen Algorithmus auf einen Computer auszuführen, müssen wir diesen als *Programm* in eine für den Computer verständliche Sprache überführen. Wir formulieren ihn dazu als in einer *Programmiersprache* mit genau spezifizierter Grammatik und Wortschatz. Wir sprechen hier vom *Quelltext*. Der Computer selbst versteht allerdings nur eine vom Prozessor (und Betriebssystem) abhängige *Maschinensprache*. Jeder Befehl ist hier einfach eine Folge einer festen Zahl von Bits (z. B. 8 oder 16 Bit bei Mikrocontrollern, 32 oder 64 Bit bei Desktop-Prozessoren). Unser Quelltext muss deshalb vor der Ausführung in die vom Prozessor verständlichen Form übersetzt werden. Dazu haben sich zwei grundlegende Mechanismen ausgebildet:

Ein *Compiler* nimmt den kompletten Quelltext her und übersetzt ihn als Einheit in Maschinencode, der wiederum in einer Datei abgelegt wird. Zur Ausführung des Programms wird nur noch diese übersetzte Datei benötigt, weder der Quelltext noch der Compiler.

Vorteile sind die im allgemeinen leichtere Verteilbarkeit und die schnellere Ausführungszeit des Programms. Nachteile sind dagegen die starke Abhängigkeit des Programms vom konkreten Prozessor und Betriebssystem (ein Programm für Windows läuft nicht unter Linux, auch wenn der gleiche PC verwendet wird) und der erhöhte Aufwand bei der Entwicklung, da nach jeder Änderung das Programm neu vom Compiler übersetzt werden muss.

Ein *Interpreter* verarbeitet den Quelltext erst zur Laufzeit und übersetzt ihn (vereinfacht gesprochen) Zeile für Zeile und führt anschließend diese Zeile sofort aus. Dazu wird der Interpreter allerdings bei jeder Programmausführung erneut benötigt.

Die Vorteile dieses Vorgehens sind die häufig kürzeren Entwicklungszeiten und die höhere Plattformunabhängigkeit: sofern für den jeweiligen Prozessor und das Betriebssystem ein Interpreter verfügbar ist, kann das Programm ohne Änderungen ausgeführt werden. Nachteile sind die meist höhere Laufzeit und die Notwendigkeit, den jeweiligen Interpreter installieren zu müssen.

Während die Programmiersprache *C* normalerweise eine compilierte Sprache ist, wird *Python* üblicherweise interpretiert.

Neben diesen beiden Ausführungsformen gibt es noch viele Mischformen. Die Programmiersprache *Java* benutzt z. B. eine *virtuelle Maschine (JVM)*, die einen ganz bestimmte, theoretische Hardwareplattform exakt beschreibt. Der Quellcode wird für diesen theoretischen Computer übersetzt. Zur Ausführung benötigen wir wiederum einen Interpreter, die virtuelle Maschine, die den Befehlssatz der theoretischen in die konkrete Hardware übersetzt. Damit erreichen wir wieder eine hohe Unabhängigkeit von der konkreten Plattform und erhalten zusätzlich bessere Laufzeiten, da die Übersetzung zwischen virtueller und realer Maschine meist sehr effizient verläuft.

Dieses Konzept wurde mit *WebAssembly (WASM)* aufgegriffen, um Programme effizient in Webbrowsern ausführen zu können. Neben der Beschleunigung der Ausführung von *JavaScript*, der meist verwendeten Browsersprache, können z. B. auch C-Programme passend für die Ausführung im Browser übersetzt werden.

1.4 Geschichte

1.4.1 C

Etwa 1970 wurde an den Bell Laboratories für den Computer PDP11 mit Unix ein neues Betriebssystem entwickelt. Um die Entwicklung zu beschleunigen und Unix auf andere Hardware zu portieren, wurde eine effiziente und hardwarenahe höhere Programmiersprache benötigt. Dennis Ritchie entwickelte in den Jahren 1971-1973 die Sprache C. Im Jahr 1978 erschien von Kernighan und Ritchie das Standardwerk [KR90].

Aus der ursprünglichen Version entstand der Standard ANSI-C (1989) und C99 (1999).

C steht auf fast allen Hardwareplattformen zur Verfügung. Durch eine Vielzahl von Bibliotheken werden praktisch auch alle Anwendungsprobleme abgedeckt.

Durch die Hardwarenähe hat C aber auch Nachteile: Es ist für Einsteiger recht fehlerträchtig und an vielen Stellen umständlich in der Nutzung. Deshalb gibt es ausgehend von C eine Vielzahl neuentwickelter Sprachen, die die Nutzung einfacher machen sollen. Am nächsten an C ist die objektorientierte Erweiterung C++, die von Bjarne Stroustrup ab 1979 entwickelt wurde.

Andere auf C basierende, aber deutlich stärker objektorientierte Sprachen sind Objective C (von Apple genutzt), Java (von Sun entwickelt, beliebt für verteilte Anwendungen) und C# (von Microsoft entwickelt, ähnlich Java).

1.4.2 Python

Python wurde Ende der 1980er Jahre von Guido van Rossum in den Niederlanden aus der Lehrsprache ABC entwickelt. Das auffälligste Merkmal der Sprache ist die notwendige korrekte Einrückung von Code-Blöcken.

Im Jahr 2000 erschien die Version 2.0, deren kompatible Nachfolge auch heute noch im Einsatz sind, obwohl der Support Ende 2019 auslief.

Seit 2008 steht die Version 3.x zur Verfügung, die gegenüber der Version 2.x eine bessere Unterstützung für Unicode bringt, aber durch die Entfernung veralteter Sprachkonstrukte zu Python 2 inkompatibel ist.

Da sich für Python relativ einfach Bibliotheken entwickeln lassen, gibt es eine Unzahl von Erweiterungen, speziell auch für die Datenanalyse und das Maschinelle Lernen, so dass diese Sprache heute in diesen Bereichen weit verbreitet ist.

1.5 Arbeitsumgebung

Für die Arbeit mit C empfiehlt sich die Verwendung von Linux, das unter Windows in einer virtuellen Box installiert werden kann. Unter *MacOSX* sollten die benötigten Werkzeuge ebenfalls verfügbar sein.

Empfehlung: [Virtual Box](#), Linux-Distribution [Debian](#), [Ubuntu](#) oder [Linux Mint](#) (für die Screencasts verwendet). Für Installation finden sich im Internet zahlreiche Anleitungen.

Unter Linux öffnen wir ein Konsolen- oder Terminalfenster (im Menü unter *Systemwerkzeuge*, in der Werkzeugleiste oder mit **AltF2** und dem Befehl **xterm**). Wir erzeugen beim ersten Mal ein Arbeitsverzeichnis für das Praktikum und wechseln in dieses:

```
# Erstellen (einmalig)
mkdir verzeichnis

# Wechsel (nach jedem Start einer Konsole)
cd verzeichnis
```

Beispiel:

- Arbeitsverzeichnis für das Praktikum: `wip1`,
- Arbeitsverzeichnis für die erste Übung: `ueb1` unter dem Verzeichnis `wip1`.

```
# Wechsel ins Home-Verzeichnis
cd
# Erstellen von wip1
mkdir wip1
# Wechsel nach wip1
cd wip1
# Erzeugen von ueb1
mkdir ueb1
# Wechsel nach ueb1
cd ueb1
# Anzeige des aktuellen Arbeitsverzeichnisses
pwd
# Ergebnis: .../wip1/ueb1
```

1.6 Wichtige Befehle der Linux-Kommandozeile

Wie wir am Beispiel gesehen haben, benötigen wir einige Befehle der Kommandozeile (der *Shell*) recht häufig zum Verwalten unserer Projekte.

cd (*ohne Parameter*) (change directory). Wechsel in das Homeverzeichnis des Nutzers, der Ausgangspunkt für alle eigenen Unterverzeichnisse.

cd *verzeichnis* Wechsel aus dem aktuellen Arbeitsverzeichnis in ein Unterverzeichnis.

cd *verzeichnis1/verzeichnis2...* Wechsel aus dem aktuellen Arbeitsverzeichnis über das Unterverzeichnis *verzeichnis1* in dessen Unterverzeichnis *verzeichnis2*.

cd *..* Wechsel aus dem aktuellen Arbeitsverzeichnis in das übergeordnete Verzeichnis.

mkdir *verzeichnis* (make directory). Erstellen eines Unterverzeichnisses im Arbeitsverzeichnis. Um hier weitere Unterverzeichnisse zu erstellen, wechseln wir vorher in dieses Verzeichnis.

pwd (print working directory). Anzeige des aktuellen Arbeitsverzeichnisses.

ls *[-l] [verzeichnis]* (list). Anzeige des Inhalts des Arbeitsverzeichnisses oder des Unterverzeichnisses *verzeichnis*.

cat *datei* (concatenate). Anzeige einer (kurzen) Quelltext- oder Textdatei im Arbeitsverzeichnis.

less *datei* (less is more). Anzeige einer (längeren) Quelltext im Arbeitsverzeichnis. Hier können wir nach vorn und hinten blättern und suchen. Beenden der Anzeige mit **q**.

cp *alteDatei neueDatei* (copy). Erzeugen einer Kopie *neueDatei* aus der alten Datei, die erhalten bleibt.

cp *alteDatei neuesVerzeichnis* (copy). Erzeugen einer Kopie mit gleichem Namen im angegebenen neuen Verzeichnis. Die Originaldatei bleibt erhalten. Nach dem Verzeichnisnamen kann auch ein neuer Dateiname angegeben werden.

mv *alteDatei neueDatei* (move). Umbenennen der alten Datei in *neueDatei*.

mv *alteDatei neuesVerzeichnis* (move). Verschieben der Datei in ein neues Verzeichnis. Nach dem Verzeichnisnamen kann auch ein neuer Dateiname angegeben werden.

1.7 Werkzeuge

Zum Erstellen von Programmen wird ein Texteditor benötigt. Empfohlen wird hier der Editor *Geany*. Wird per SSH auf einem PC der TU gearbeitet, ist die Verwendung eines graphischen Editors nicht möglich. Hier kann als einfach zu bedienender Editor *nano* verwendet werden.

C-Programme müssen anschließend in Maschinencode übersetzt werden, dazu dient ein Compiler. Wir verwenden entweder den *GNU*-Compiler **gcc** oder den *LLVM*-Compiler **clang**. Beide unterstützen die gleichen Operationen.

Für Python benutzen wir den Python-3-Interpreter **python3**.

1.8 Testen der Umgebung

Wir testen zunächst das Erstellen, Übersetzen und Ausführen eines C-Programms. Dazu wechseln wir in unsere Arbeitsumgebung und erstellen mittels **geany** den Quelltext **t1.c**.

```
geany t1.c &
```

Das &-Zeichen koppelt das Editorfenster von unserer Konsole ab, so dass wir hier weiter Befehle ausführen können. Vergessen wir es, ist die Konsole solange blockiert, bis der Editor beendet wird.

Wir geben nun den Quelltext des Testprogramms ein (auf korrekte Groß- und Kleinschreibung sowie auf Leerzeichen achten).

Quellcode 1.1: t1.c

```
#include <stdio.h>

int main() {
    printf("Hallo\n");
}
```

Download zum Vergleich: **t1.c**.

Wir speichern den Quelltext mit **StrgS** und überzeugen uns durch die Verzeichnisanzeige, dass die Datei vorhanden ist:

```
ls
```

Wir können zur Kontrolle diese Datei auch anzeigen:

```
cat t1.c
```

Der Quelltext muss nun übersetzt werden:

```
# GNU-C
gcc -std=c99 -Wall -o ziel quelle.c

# oder mit LLVM-C
clang -std=c99 -Wall -o ziel quelle.c
```

Bemerkung: Der LLVM-Compiler liefert häufig ausführlichere Warnungen, so dass es sinnvoll ist, beide Compiler für die Suche nach potentiellen Problemen zu nutzen.

Es entsteht eine neue Datei mit dem Namen des Ziels. Wir überzeugen uns davon mit dem Kommando **ls**:

```
ls -l
```

und erhalten z. B.

```
insgesamt 24
-rwxrwxr-x 1 jens jens 16688 Aug 19 07:46 t1
-rw-rw-r-- 1 jens jens   54 Aug 19 07:45 t1.c
```

Das Zeichen `x` im Block der ersten Spalte zeigt uns, dass es sich bei `t1` um ein ausführbares Programm handelt. Wir können dieses nun starten:

```
./t1
```

Bemerkung: Aus Sicherheitsgründen werden Programme nur dann automatisch gestartet, wenn sie sich in speziellen, schreibgeschützten Verzeichnissen befinden. Um ein Programm aus einem eigenen Verzeichnis zu starten, muss dieses Verzeichnis explizit angegeben werden. Für unser aktuelles Arbeitsverzeichnis geschieht dies durch Voranstellen von `./`.

Zum Test der Python-Umgebung starten wir einfach den Interpreter:

```
python3
```

Wir erhalten etwa folgende Ausgabe:

```
Python 3.8.2 (default, Jul 16 2020, 14:00:26)
[GCC 9.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Zum Beenden geben wir ein:

```
quit()
```

oder drücken einfach die Tastenkombination **StrgD**.

Wollen wir in Python Programme schreiben, erstellen wir ebenso wie für C mit einem Editor eine Quelltextdatei, verwenden nun aber die Endung `.py`. Beispiel:

Quellcode 1.2: `t1.py`

```
print('Hallo')
```

Dieses Programm können wir nun dem Python-Interpreter beim Aufruf mitgeben:

```
python3 t1.py
```

Ein Übersetzen ist nicht erforderlich. Nach Beendigung des Programms wird im Gegensatz zum interaktiven Python-Start auch der Interpreter automatisch wieder beendet.

1.9 Aufgaben

Aufgabe 1.1 Erstellen Sie ein Verzeichnis für eine Hausaufgabe `:file:'ha'` im Verzeichnis `:file:'wip1'`.

Erzeugen Sie hier einen C-Quelltext `pp.c`, der die Ausgabe

Programmieren macht Spaß!

erzeugt. Übersetzen und testen Sie Ihr Programm.

Inhalt

- *Aufbau von C-Programmen*
- *Übersetzung*
- *Variablen und Ausdrücke*
- *Kommentare*
- *Ausgabe*
- *Eingabe*
- *Konstanten*
- *Bezeichner*
- *Literale*
- *Aufgaben*



Eine ausführliche und gut lesbare Beschreibung der Sprache bietet das Buch [KR90] von zwei der C-Entwickler. Im URZ-Nutzerservice ist die recht preiswerte Anleitung [SPR.C1] erhältlich. Eine recht umfangreiche Dokumentation stellt [Wolf20] dar, wovon der Verlag freundlicherweise eine kostenlose Online-Version bereitstellt. Im Gegensatz zum vorigen Buch wendet sich [WoKr20] an Einsteiger und erläutert die C-Programmierung systematisch.

2.1 Aufbau von C-Programmen

Wir betrachten wieder unser erstes Programm `t1.c`:

Quellcode 2.1: Minimalprogramm

```
#include <stdio.h>

int main() {
    printf("Hallo\n");
}
```

Es hat folgende Bestandteile:

#include <stdio.h>

Da wir für die Ausgabe eine Funktion benötigen, die wir nicht selbst definieren, müssen wir dem Compiler mitteilen, wie diese zu benutzen ist. Das kann entweder durch eine eigene Deklaration oder durch die Einbindung einer vordefinierten Deklaration geschehen. Dieses Makro sammelt diese für alle gängigen Ein- und Ausgabefunktionen.

int main()

Hauptfunktion eines C-Programms. C-Programme bestehen aus einer Zahl von Funktionen, die aufgerufen werden und einen Wert zurückgeben. Die **main**-Funktion gibt beim Beenden automatisch einen Wert an das Betriebssystem zurück, der diesem mitteilt, ob das Programm korrekt gearbeitet hat.

printf("text")

Funktionsruf, Ausgabe der übergebenen Werte auf der Konsole.

Die Anweisung **printf()**; ist in geschweifte Klammern eingeschlossen. Diese bilden einen *Block* zusammengehöriger Anweisungen. Eine solche Blockbildung ist für die Anweisungen einer Funktion und immer dann erforderlich, wenn die C-Syntax an dieser Stelle nur eine Anweisung erlaubt.

Bemerkung: Um Blöcke gut lesbar zu gestalten, sollten alle Anweisungen eines Blocks mit der gleichen Zahl von Leerzeichen eingerückt werden (empfohlen: 2 oder 4). Die Klammern selbst werden nicht eingerückt, wobei die öffnende Klammer häufig auf die vorige Zeile gezogen wird, die den Block einleitet:

```
anweisungMitEinerUnteranweisung {
    Anweisung1;
    Anweisung2;
    ...
}
```

Allgemein bestehen C-Programme aus einer Folge von Funktionsdefinitionen mit folgendem Aufbau:

```
rückgabeTyp funktionsName(...) {
    anweisung;
    ...
}

...
```

2.2 Übersetzung

Wie wir bereits gesehen haben, müssen wir unseren Quelltext in Maschinencode übersetzen, ehe wir es ausführen können. Wir benutzen dazu einen Compiler, entweder **gcc** für den *GNU*-Compiler oder **clang** für den *LLVM*-Compiler.

Der kürzeste Aufruf ist

```
gcc quelltext.c

# oder

clang quelltext.c
```

Es entsteht jeweils eine Datei `a.out`. Dieser Name wurde historisch festgelegt und aus Kompatibilitätsgründen nie geändert. Da dies mit jedem neuen Programm das bisherige Programm überschreiben würde, wollen wir den Programmnamen selbst festlegen. Weiter ist es sinnvoll, den zu benutzenden C-Standard genau festzulegen und potentielle Fehlerquellen anzuzeigen. Dies wird über *Optionen* oder *Kommandozeilenparameter* spezifiziert. Die wichtigsten Optionen für uns sind:

- o **programmdatei** Festlegung des Namens der Programmdatei.
- std=C-Standard Festlegung des benutzten C-Standards. Beide Compiler benutzen standardmäßig eigene, compilerspezifische Erweiterungen, die sich von Version zu Version unterscheiden. Wir legen uns deshalb auf den Wert `c99` fest.
- Wall Einschalten von Warnungen bei potentiellen Fehlerquellen
- Wextra Einschalten von noch mehr Warnungen bei potentiellen Fehlerquellen
- Werror Abbruch der Übersetzung beim Auftreten von Warnungen
- g Übersetzen im Debug-Modus. Damit können wir zur Fehlersuche das Programm schrittweise ausführen.

Die Reihenfolge der Optionen ist nicht festgelegt, allerdings muss der Parameter einer Option direkt hinter dieser stehen (z. B. `-o programmdatei`).

Unser Compileraufruf sollte also normalerweise so aussehen:

```
gcc -o programmdatei -std=c99 -Wall quelldatei.c
```

2.3 Variablen und Ausdrücke

Um in unserem Programm Algorithmen für unterschiedliche Werte ausführen zu können, benötigen wir die Möglichkeit, entsprechende Speicherzellen zu definieren, in denen wir diese Werte ablegen können. Diese bezeichnen wir als *Variablen*. Als Beispiel wollen wir aus Geschwindigkeit und Zeit die zurückgelegte Strecke berechnen.

```
int zeit = 30;
int geschwindigkeit = 8;
int strecke;
```

Wir definieren damit die Variablen `zeit`, `geschwindigkeit` und `strecke` speichern in ihnen die Werte 30 bzw. 8. Die dritte Variable wird dagegen nicht *initialisiert* und enthält einen unbestimmten Wert. Wir benutzen sie, um hier später einen berechneten Wert abzulegen.

Bemerkung: Jede Variable muss vor ihrer Verwendung *definiert* werden.

Die Angabe `int` bestimmt den *Typ* einer Variablen und darf nicht fehlen. Der Typ legt fest, wieviel Speicher zum Speichern des Wertes vorgesehen ist und wie die Rechenregeln für diese Variablen aussehen. Mit dem Typ `int` können wir ganze Zahlen speichern.

Wir wollen nun die Strecke nach der bekannten Formel $s = v \cdot t$ berechnen:

```
strecke = geschwindigkeit * zeit;
```

Wir schreiben hier ein `=`-Zeichen, meinen aber eine *Zuweisung*: Berechne `geschwindigkeit * zeit` und speichere das Ergebnis in der Variablen `strecke`:

```
strecke <- geschwindigkeit · zeit
```

Zuweisungen haben folgenden Aufbau:

```
variable = ausdruck
```

Der Ausdruck rechts vom Gleichheitszeichen wird ausgewertet und dieser Wert der Variablen zugewiesen. Die Zuweisung selbst hat wieder einen Wert, der als Ausdruck verwendet werden kann, es sind also Zuweisungsketten möglich:

```
a = b = c = 0;
```

Normalerweise wird die Zuweisung als Anweisung benutzt, müssen wir sie mit einem Semikolon abschließen.

Für Rechnungen (korrekt: *arithmetische Ausdrücke*) stehen uns die Operatoren `+`, `-`, `*` für die Multiplikation und `/` für die (ganzzahlige) Division zur Verfügung. Zusätzlich können wir noch den Divisionsrest mit dem Operator `%` bestimmen. Punktrechnung geht vor Strichrechnung, die Berechnung erfolgt von links nach rechts. Wollen wir die Reihenfolge beeinflussen, können wir Teilausdrücke klammern. Beispiel:

```
int a = 7;
int b = 3;
int c;

c = (a - 1 - b) * (a + b) % 7;
```

Es wird der Reihe nach berechnet:

```
a - 1 - b → 3
a + b     → 10
3 * 10    → 30
30 % 7    → 2, da 30 = 4 · 7 + 2
```

Bemerkung: Für das Klammern von Teilausdrücken dürfen nur die runden Klammern `()` verwendet werden. Eckige und geschweifte Klammern haben eine andere Bedeutung. Klammern dürfen verschachtelt werden, jedoch muss jede öffnende Klammer wieder geschlossen werden.

Um unser Programm zur Geschwindigkeitsberechnung abzuschließen, benötigen wir noch eine Möglichkeit, den Inhalt von Variablen auszugeben. Ebenso wollen wir unser Programm generalisieren und die Berechnung für verschiedene Zeiten und Geschwindigkeiten ausführen können.

2.4 Kommentare

In unserem Beispiel benutzen wir die Variablen `geschwindigkeit`, `zeit` und `strecke`, die alle irgendwelche Zahlen enthalten. Wir wissen aber, dass wir unsere Formel nur dann anwenden können, wenn auch die jeweiligen Einheiten zueinander passen. Bei uns ist jedoch völlig unklar, welche Einheiten wir verwenden wollen. Um dies zu dokumentieren, haben wir zwei Möglichkeiten:

- Verwenden aussagekräftiger Variablennamen, z. B. `strecke_m` für die Strecke in Meter und `geschw_m_s` für die Geschwindigkeit in Meter pro Sekunde.
- Verwendung von Kommentaren.

Soweit möglich, sollte die erste Variante bevorzugt werden. Oft sind jedoch längere Erklärungen möglich. Dafür verwenden wir *Kommentare*:

Einzeilige Kommentare Sie werden mit `//` eingeleitet und wirken bis zum Zeilenende

Kommentarblöcke Sie werden mit `/*` eingeleitet und bis zum ersten Auftreten von `*/` auch über mehrere Zeilen fortgesetzt.

Um also unsere Variablen zu erklären, könnten wir wir folgt vorgehen:

```
/* Variablen für Geschwindigkeitsberechnung
 * (Dieser Stern ist nur Verzierung)
 * Offensichtliche Dinge wie diese sollten nicht kommentiert werden!
 */
int zeit = 30;           // in Sekunden
int geschwindigkeit = 8; // in Meter pro Sekunde
int strecke;            // in Metern
```

2.5 Ausgabe

Für die flexible Ausgabe von Beschreibungen (*Strings*), numerischen Werten, Variablen und Ausdrücken verwenden wir die Funktion `printf()`. Diese erwartet als ersten Parameter ein *Muster* mit *Platzhaltern*, die durch die folgenden Parameter ersetzt werden. Es muss genauso viele Platzhalter geben, wie Parameter folgen. Wir benötigen zunächst nur Platzhalter für ganze Zahlen, die in der einfachsten Form mit `%d` dargestellt werden.

```
printf("Text %d\n", intAusdruck);
```

Verwenden wir den Platzhalter mehrfach, müssen wir entsprechend weitere ganzzahlige Ausdrücke als Parameter übergeben.

```
printf("Zeit %d s, Geschw. %d m/s, Strecke %d m\n",
      zeit, geschwindigkeit, strecke);
```

Das abschließende Zeichen `\n` sorgt dafür, dass nach der Ausgabe die Zeile beendet wird und die nächste Ausgabe auf einer neuen Zeile startet.

Wir können nun eine erste Version unseres Programms komplett erstellen und testen.

Quellcode 2.2: Geschwindigkeitsberechnung, Version 1

```
/*
 * Streckenberechnung bei gleichförmiger Bewegung.
 */

#include <stdio.h>

int main() {
    int zeit = 30;           // in Sekunden
    int geschwindigkeit = 8; // in Meter pro Sekunde
    int strecke;            // in Metern

    strecke = zeit * geschwindigkeit;
    printf("Zeit %d s, Geschw. %d m/s, Strecke %d m\n",
          zeit, geschwindigkeit, strecke);
}
```

Wollen wir verschieden große Zahlen in einer Tabelle gleich ausrichten, dann ist es wünschenswert, die alle Zahlen auf gleiche Länge mit Leerzeichen aufzufüllen. Dies geschieht durch Einschieben der Längenangabe in das Formatierungssymbol zwischen `%` und `d`.

Beispiel: Einige Quadratzahlen formatiert ausgeben:

```
#include <stdio.h>

/* Demonstration Formatierung mit Ausrichtung */

int main() {
    int x;
    x = 5;
    printf("%2d^2 = %4d\n", x, x*x);
    x = 21;
    printf("%2d^2 = %4d\n", x, x*x);
    x = 9;
    printf("%2d^2 = %4d\n", x, x*x);
}
```

Ergebnis:

```
5^2 =   25
21^2 =  441
9^2 =   81
```

Wollen wir das Prozentzeichen selbst ausgeben, müssen wir es im Formatstring doppelt schreiben: **%%**.

Um unsere Programmausgaben dauerhaft aufbewahren, können wir die Ausgabe in eine Datei umlenken:

```
./programm > ausgabedatei
```

Als Dateierweiterung verwenden wir sinnvollerweise **txt** oder **out**. Die Ausgabe des obigen Beispiels wurde durch folgenden Aufruf erzeugt, um sie ins Script einbinden zu können:

```
./intformat > intformat.out
```

2.6 Eingabe

Damit unsere Programme nicht jedesmal für neue Daten neu übersetzt werden müssen, benötigen wir eine Möglichkeit, diese Werte während der Laufzeit über eine *Eingabe* lesen zu können. Wir betrachten dazu zwei Möglichkeiten.

Beim Aufruf eines Programms besteht die Möglichkeit, diesem Werte in Form von Parametern mitzugeben:

```
./programm [parameter ...]
```

Die Parameter werden durch Leerzeichen getrennt und müssen in einer Zeile stehen. Um diese zu lesen, müssen wir die *main()*-Funktion modifizieren:

```
int main(int argc, char * argv[]) {
    ...
}
```

Die Variable *argc* enthält nun die Anzahl der Parameter (einschließlich des Programmnamens!), die Variable *argv* die tatsächlichen Werte des Programms als *Feld*. Wir müssen diesen Mechanismus hier nicht verstehen, sondern verwenden einfach folgendes Muster, um unserem Programm Werte mitzugeben:

```
#include <stdlib.h>

int main(int argc, char* argv[]) {
    int parameter1 = atoi(argv[1]);
    int parameter2 = atoi(argv[2]);
}
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
// ...
}
```

Übergeben wir weniger Parameter, als wir im Programm lesen wollen, oder lassen sich diese nicht als ganze Zahlen interpretieren, stürzt das Programm ab, oder unsere Variablen erhalten undefinierte Werte. Wir sollten deshalb zur Kontrolle unbedingt die gelesenen Werte mit ausgeben.

Beispiel: Geschwindigkeitsberechnung mit Lesen der Werte von der Kommandozeile.

Quellcode 2.3: geschw1.c

```
/*
 * Streckenberechnung bei gleichförmiger Bewegung.
 */

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

int main(int argc, char *argv[]) {
    int zeit;           // in Sekunden
    int geschwindigkeit; // in Meter pro Sekunde
    int strecke;        // in Metern

    zeit = atoi(argv[1]);
    geschwindigkeit = atoi(argv[2]);
    strecke = zeit * geschwindigkeit;
    printf("Zeit %d s, Geschw. %d m/s, Strecke %d m\n",
           zeit, geschwindigkeit, strecke);
}
```

Beim Aufruf des Programms mit

```
./geschw1 20 8
```

erhalten wir

```
Zeit 20 s, Geschw. 8 m/s, Strecke 160 m
```

Statt die Werte direkt in der Kommandozeile mitzugeben, kann unser Programm sie auch bei Bedarf zur Laufzeit anfordern. Zum Lesen von ganzen Zahlen benutzen wir dazu folgende Funktion, die wir später verstehen werden:

```
#include <stdio.h>

int readInt() {
    int res;
    scanf("%d", &res);
    return res;
}
```

Download: `readInt.inc`

Wir kopieren diesen Code einfach an den Anfang unserer Programmdatei. Nun können wir einen Wert mit folgender Anweisung lesen:

```
int variable = readInt();
```

Als Beispiel betrachten wir wieder unsere Geschwindigkeitsberechnung.

Quellcode 2.4: geschw2.c Lesen der Eingabedaten von der Konsole

```
/*
 * Streckenberechnung bei gleichförmiger Bewegung.
 * Eingabe über Standardeingabe bzw. aus Datei mittels
 * ./geschw2 < datei
 */

#include <stdio.h>

/*
 * Lesen eines int-Wertes von der Standardeingabe
 */

int readInt() {
    int val = 0;
    scanf("%d", &val);
    return val;
}

int main() {
    int zeit;           // in Sekunden
    int geschwindigkeit; // in Meter pro Sekunde
    int strecke;        // in Metern

    zeit = readInt();
    geschwindigkeit = readInt();
    strecke = zeit * geschwindigkeit;
    printf("Zeit %d s, Geschw. %d m/s, Strecke %d m\n",
        zeit, geschwindigkeit, strecke);
}
```

Wir rufen dieses Programm ohne Paramter auf:

```
./geschw2
```

Das Programm bleibt nun stehen und wartet auf Eingaben. Diese geben wir nun durch Leerzeichen getrennt in einer neuen Zeile ein:

```
42 23
```

und erhalten:

```
Zeit 42 s, Geschw. 23 m/s, Strecke 966 m
```

Diese zweite Variante erlaubt uns das Lesen der Daten aus einer Datei, in der wir die Werte speichern, was insbesondere bei einer größeren Zahl von Daten das Testen erleichtert.

Wir schreiben dazu unsere Daten durch Leerzeichen oder Zeilenschaltung getrennt in eine Textdatei data1.txt:

Quellcode 2.5: data1.txt

```
19 47
```

Unser Programm wird wie folgt aufgerufen:

```
./geschw2 < data1.txt
```

Das Kleiner-Zeichen ist eine sogenannte *Datei umlenkung*. Statt von der Konsole wird nun auf gleiche Weise aus der angegebenen Datei gelesen.

Wollen wir gleichzeitig aus einer Datei lesen und die Ergebnisse in einer Datei speichern, können wir auch die Ausgabe umlenken:

```
./geschw2 < data1.txt > data1.out
```

Bei der interaktiven Nutzung mit der Eingabe der Werte von der Konsole sehen wir allerdings keinerlei Hinweise, was einzugeben ist. Um diese anzuzeigen, fügen wir einfach vor der Eingabe eine Textausgabe mit dem Hinweis ein.

Quellcode 2.6: geschw3.c Eingabe mit Hinweisen

```
/*
 * Streckenberechnung bei gleichförmiger Bewegung.
 * Eingabe über Standardeingabe mit Hinweisen.
 */

#include <stdio.h>

/*
 * Lesen eines int-Wertes von der Standardeingabe
 */

int readInt() {
    int val = 0;
    scanf("%d", &val);
    return val;
}

int main() {
    int zeit;           // in Sekunden
    int geschwindigkeit; // in Meter pro Sekunde
    int strecke;         // in Metern

    printf("Zeit [s]:");
    zeit = readInt();
    printf("Geschwindigkeit [m/s]:");
    geschwindigkeit = readInt();
    strecke = zeit * geschwindigkeit;
    printf("Zeit %d s, Geschw. %d m/s, Strecke %d m\n",
           zeit, geschwindigkeit, strecke);
}
```

Die Ausgabe der Hinweise für die Eingabe ist beim Einlesen der Daten aus einer Datei allerdings nutzlos und verwirrt nur, insbesondere, wenn das Ergebnis ebenfalls in einer Datei gespeichert werden soll. Wir sollten diese also bei Programmen für die automatische Verarbeitung von Datendateien besser weglassen.

2.7 Konstanten

Oft benötigen wir in unseren Programmen feste Werte, die während der Laufzeit nicht geändert werden sollen, wie die Dauer einer Semesterwochenstunde (Unterrichtsstunde) in Minuten oder die Zahl der Wochen eines Jahres. Da die direkte Verwendung von Zahlenliteralen das Programm schlechter lesbar macht oder sich manche Werte auch ändern können, was zu vielen notwendigen Korrekturen im Programm führen würde, bietet sich die Verwendung von *Konstanten* an.

```
const typ konstantenName = wert;
```

Konstanten sollten ganz am Anfang außerhalb aller Funktionen definiert werden, damit sie überall verfügbar sind. Für Konstantennamen empfiehlt sich die Schreibung in Großbuchstaben. Beispiele:

```
const int WEEKS_YEAR = 52; // Wochen pro Jahr
const int LECTHOUR_MIN = 45; // Dauer Vorlesungsstunde
```

Beispiel: Umrechnung der Geschwindigkeit in km/h und der Fahrzeit in Stunden/Minuten/Sekunden in der Ausgabe des Geschwindigkeitsprogramms.

Quellcode 2.7: geschw4.c: Umrechnung in km/h, H/Min/Sek in der Ausgabe mit Konstanten

```
/*
 * Streckenberechnung bei gleichförmiger Bewegung.
 * Eingabe über Standardeingabe bzw. aus Datei mittels
 * ./geschw2 < datei
 * Umrechnung in km/h mit Konstante
 */

#include <stdio.h>

const int SEC_MIN = 60;      // Sek. pro Min.
const int MIN_HOUR = 60;    // Min. pro Stunde
const int KM_H_FACTOR = 360; // in %

/*
 * Lesen eines int-Wertes von der Standardeingabe
 */

int readInt() {
    int val = 0;
    scanf("%d", &val);
    return val;
}

int main() {
    int zeit;           // in Sekunden
    int geschwindigkeit; // in Meter pro Sekunde
    int strecke;        // in Metern

    zeit = readInt();
    int min = zeit / SEC_MIN;
    int sec = zeit % SEC_MIN;
    int hour = min / MIN_HOUR;
    min = min % MIN_HOUR; // min wird überschrieben!
    geschwindigkeit = readInt();
    int kmH = KM_H_FACTOR * geschwindigkeit / 100;
    strecke = zeit * geschwindigkeit;
    printf("Zeit %d h %d min %d s, Geschw. %d km/h, Strecke %d m\n",
        hour, min, sec, kmH, strecke);
}
```

Eine zweite Möglichkeit, Konstanten zu vereinbaren, ist die Verwendung des `enum`-Datentyps.

```
enum {KONSTANTE = wert [, ...]};
```

Im Gegensatz zu obigen Konstanten, die sich wie unveränderliche Variablen verhalten, werden `enum`-Konstanten vom Compiler durch die entsprechenden Literale ersetzt. Bei der Vereinbarung von `Feldern` benötigen wir die zweite Form. Mit ihr können wir allerdings nur ganzzahlige Konstanten definieren, aber keine reellen Zahlen oder Texte.

Quellcode 2.8: geschw5.c: Umrechnung in km/h, H/Min/Sek in der Ausgabe mit Konstanten als enum definiert

```

/*
 * Streckenberechnung bei gleichförmiger Bewegung.
 * Eingabe über Standardeingabe bzw. aus Datei mittels
 * ./geschw2 < datei
 * Umrechnung in km/h mit Konstante
 */

#include <stdio.h>

enum {SEC_MIN = 60,          // Sek. pro Min.
      MIN_HOUR = 60,        // Min. pro Stunde
      KM_H_FACTOR = 360     // in %
};

/*
 * Lesen eines int-Wertes von der Standardeingabe
 */

int readInt() {
    int val = 0;
    scanf("%d", &val);
    return val;
}

int main() {
    int zeit;                // in Sekunden
    int geschwindigkeit;     // in Meter pro Sekunde
    int strecke;              // in Metern

    zeit = readInt();
    int min = zeit / SEC_MIN;
    int sec = zeit % SEC_MIN;
    int hour = min / MIN_HOUR;
    min = min % MIN_HOUR;    // min wird überschrieben!
    geschwindigkeit = readInt();
    int kmH = KM_H_FACTOR * geschwindigkeit / 100;
    strecke = zeit * geschwindigkeit;
    printf("Zeit %d h %d min %d s, Geschw. %d km/h, Strecke %d m\n",
           hour, min, sec, kmH, strecke);
}

```

Auf eine dritte Form der Konstantenvereinbarung über Präprozessor-Makros wollen wir hier nicht eingehen.

2.8 Bezeichner

Um selbstdefinierte Konstrukte wie Variablen, Funktionen, Datentypen, ansprechen zu können, müssen diese eindeutige Namen erhalten. Für diese sogenannten Bezeichner (engl. identifier) gelten folgende Regeln:

- Sie bestehen aus den lateinischen Buchstaben A...Z (groß), a...z (klein), Ziffern und dem Sonderzeichen `_`.
- Umlaute (ä, ö, ü, ß), Leerzeichen und Sonderzeichen sind nicht zulässig.

- Das erste Zeichen ist immer ein Buchstabe (groß oder klein).
- Die Folgezeichen sind beliebig, die Länge ist ebenfalls beliebig.
- Bezeichner müssen innerhalb ihres Deklarationsbereichs (*scope*) eindeutig sein.
- Groß- und Kleinschreibung wird unterschieden: `menge`, `Menge` und `MENGE` sind unterschiedliche Bezeichner und können gleichzeitig für verschiedene Objekte benutzt werden.
- Bezeichner dürfen nicht mit Schlüsselwörtern der Sprache C übereinstimmen.

Bezeichner sollten «sprechende Namen» haben und damit beschreiben, wofür sie stehen. Werden dabei Wörter zusammengesetzt, bieten sich folgende Formen an:

- mit Unterstrich: `velocity_km_h`
- *KamelSchreibWeise* (CamelCase), neue Teilwörter beginnen mit einem Großbuchstaben: `velocityKmH`

Es ist in C Konvention, dass Bezeichner für Variablen und Funktionen mit einem Kleinbuchstaben beginnen. Bezeichner, die mit einem Großbuchstaben beginnen, können sinnvoll für selbstdefinierte Typen und Strukturen (in Python und Java für Klassen) verwendet werden. Für Konstanten wie z. B. die größtmögliche Zahl des eingebauten Zufallsgenerators oder $\pi = 3.1415926\dots$ werden Bezeichner verwendet, die nur aus Großbuchstaben bestehen: `RAND_MAX`, `M_PI`.

2.9 Literale

Neben Bezeichnern benötigen wir oft noch feste Werte wie bestimmte Zahlen, sogenannte Literale. Zunächst benötigen wir nur ganze Zahlen und feste Texte oder *Strings* für die Ausgabe.

Ganze Zahlen werden wie üblich im Dezimalsystem geschrieben und dürfen ein `-` als Vorzeichen haben. Leerzeichen oder Gruppentrenner sind nicht zulässig.

Beispiele für Integer-Literale:

```
42
-23
0
12345
```

Bemerkung: Es gibt spezielle Integer-Literale zur Darstellung von Zahlen im Hexadezimal- und im Oktalsystem: Beginnt das Literal mit einer 0, haben wir es mit einem Oktalwert zu tun, beginnt es mit `0x`, dann handelt es sich um einen Hexadezimalwert:

```
42 == 0x2a == 052
```

Ein Zahlenliteral wie `089` ist ungültig, da 8 und 9 keine Oktalziffern sind.

Strings werden in doppelte Anführungszeichen (`"`) eingeschlossen. Der eigentliche Stringwert enthält diese Zeichen nicht, sind sind nur eine Information an den C-Compiler, wie der Wert intern zu speichern und zu verwenden ist. Innerhalb des Strings können alle *ASCII*-Zeichen verwendet werden – die deutschen Umlaute gehören nicht dazu.

Für einige Zeichen wie das Begrenzungszeichen benötigen wir eine Maskierung:

`\"` Anführungszeichen.

`\'` Apostroph, kann hier auch direkt geschrieben werden.

`\n` Zeilenschaltung.

`\t` Tabulator.

\\ Das *Backslash*-Zeichen \ selbst.

\xhexZahl Hexadezimalwert (zweistellig!) des Zeichens, falls nicht direkt auf der Tastatur vorhanden.

Beispiele:

```
"Hallo C!\n"
"\x2a\x02"
""Python" macht Spass ...?\n"
```

Bemerkung: Wenn wir Umlaute oder das ß in Stringliteralen verwenden, wird unser Programm trotzdem funktionieren. Sobald wir es aber auf einem anderen Betriebssystem übersetzen oder den String im Programm verarbeiten wollen, erhalten wir merkwürdige Effekte.

Wir sollten deshalb in C auf die Verwendung von Umlauten komplett verzichten.

2.10 Aufgaben

Aufgabe 2.1 Erstellen Sie ein erstes C-Programm, übersetzen und testen Sie es.

Aufgabe 2.2 Schreiben Sie ein C-Programm zum Testen des oben aufgeführten *Klammerausdrucks*.

Aufgabe 2.3 Schreiben Sie ein Programm, das zwei Zahlen von der Kommandozeile liest und diese in den Variablen **a** und **b** speichert. Anschließend soll der Inhalt der beiden Variablen vertauscht werden. Geben Sie den Variableninhalt zur Kontrolle vor und nach der Vertauschung aus.

Aufgabe 2.4 Schreiben Sie ein Programm zum Lösen der Gleichung $a \cdot x + b = c$. Die drei ganzen Zahlen sollen aus einer Datei (bzw. von der Konsole) gelesen werden. Die Ausgabe soll als gemischter Bruch erfolgen. Beispiel: Für die Werte 3, 2 7 erhalten wir als Ergebnis $1\frac{2}{3}$, das wir so darstellen:

```
3 * x + 2 = 7
x = 1_2/3
```

Einfache Funktionen

- *Funktionsbegriff*
- *Definition von Funktionen*
- *Funktionen ohne Rückgabewert*
- *Einfache Verzweigungen*
- *Zusicherungen*
- *Rekursive Funktionen*
- *Aufgaben*

3.1 Funktionsbegriff

Eine *Funktion* ordnet einem *Element* (Wert) aus dem *Definitionsbereich* der Funktion *eindeutig* einen *Wert* aus dem *Wertebereich* zu. In C und wird der Funktionsbegriff etwas allgemeiner verwendet, wir wollen jedoch zunächst nur Funktionen betrachten, die dieser Bedingung genügen.

Bisher kennen wir nur den Datentyp `int`. Unser Definitionsbereich sind also ganze Zahlen, eine Teilmenge davon oder *Tupel* ganzer Zahlen. So kann der Definitionsbereich z. B. die Menge aller Paare positiver ganzer Zahlen sein, die die Kantenlängen von Rechtecken repräsentieren, und unser Wertebereich sind positive ganze Zahlen für die Fläche oder den Umfang des Rechtecks.

In C und Python können wir als Definitions- und Wertebereich einer Funktion nur den kompletten Definitionsbereich eines entsprechenden Datentyps angeben, eine Beschränkung auf Teilmengen ist nicht möglich.

Zur Veranschaulichung bietet sich das Modell der kleinen Männchen (kurz Zwerge) an. Ein Zwerg erhält Anweisungen (den Algorithmus) und die Werte/Parameter, für die er die Anweisungen ausführt. Dafür benutzt er eigene Ressourcen (Notizen == Variablen, Nachschlagewerke = Hilfsfunktionen), von denen der Auftraggeber nichts erfährt. Hat er die Bearbeitung abgeschlossen, gibt er das Ergebnis an den Auftraggeber zurück.

3.2 Definition von Funktionen

In C wird eine Funktion wie folgt definiert:

```
rückgabetyf funktionsName(parameter[, parameter ...]) {
    anweisung;
    ...
}
```

Der Rückgabetyf ist für uns zunächst nur `int`. Der Funktionsname selbst ist ein *Bezeichner*. Die Parameterliste ist eine durch Komma getrennte Liste der einzelnen Parameter, die wie folgt angegeben werden:

```
typ parameterName
```

wobei der Parametername wiederum ein Bezeichner ist.

Da die Funktion einen Wert zurückliefern soll, müssen wir diesen spezifizieren:

```
return ausdruck;
```

Mit der `return`-Anweisung wird die Funktion beendet. Nachfolgende Anweisungen werden nicht ausgeführt.

Beispiel für die Flächenberechnung des Rechtecks:

```
/* Flächenberechnung eines Rechtecks.
 * a, b: Seitenlängen
 */
int rectArea(int a, int b) {
    int ar = a * b;
    return area;
}
```

Um diese Funktion zu nutzen, müssen wir sie mit konkreten Werten aufrufen. Wir schreiben uns z. B. eine `main()`-Funktion, definieren die beiden Seitenlängen und berechnen die Fläche.

Quellcode 3.1: Flächenberechnung Rechteck und einfacher Test

```
#include <stdio.h>

/* Flächenberechnung eines Rechtecks.
 * a, b: Seitenlängen
 */
int rectArea(int a, int b) {
    int area = a * b;
    return area;
}

int main() {
    int a = 3;
    int b = 5;
    int area = rectArea(a, b);
    printf("Rechteckflaeche bei Seitenlaengen %d und %d ist %d.\n",
        a, b, area);
}
```

Download: `area1.c`.

Die übergebenen Parameter der Funktion verhalten sich wie normale Variable. Allerdings wird ihnen beim Aufruf der Funktion ein Wert zugewiesen. Nach dem Verlassen der Funktion gehen sie automatisch

verloren. Auch wenn beim Aufruf der Funktion die gleichen Variablennamen benutzt werden, sind die Funktionsparameter trotzdem neue Variablen, die die ursprünglichen Variablen nicht beeinflussen.

3.3 Funktionen ohne Rückgabewert

Viele Programmiersprachen benutzen die Funktionssyntax auch für in sich abgeschlossene Codeabschnitte, die keine Berechnung ausführen, sondern mit dem Betriebssystem kommunizieren, um z. B. eine Bildschirmausgabe zu produzieren. Dies wird durch den speziellen Rückgabebetyp `void` in der Funktionsdefinition angegeben. Benutzt die Funktion eine `return`-Anweisung, darf hinter ihr kein Ausdruck stehen.

Beispiel: Ausgabe der Seitenlängen und berechneten Werte unseres Rechtecks.

```
/* Anzeige der Rechteckeigenschaften.
 * a, b: Seitenlängen
 */
void showRectangle(int a, int b) {
    int area = rectArea(a, b);
    int peri = rectPerimeter(a, b);
    printf("Rechteck mit Seitenlaengen: %d %d\n", a, b);
    printf("  Flaeche: %d, Umfang %d\n", area, peri);
    return; // kann entfallen
}
```

Bemerkung: Funktionen für Berechnungen und Ausgaben sollten im Programm klar getrennt werden. Das erleichtert zum einen das Testen der Funktionen, wenn diese weniger komplex sind, und zum anderen die Wiederverwendung der Berechnung, wenn an dieser Stelle gar keine oder eine andere Form der Ausgabe (z. B. Speichern in einer Datenbank) erforderlich ist.

3.4 Einfache Verzweigungen

Sehr oft ist es erforderlich, in Abhängigkeit von eingelesenen oder berechneten Werten im Algorithmus unterschiedliche Aktionen auszuführen. So sollten z. B. bei der automatischen Erfassung von Wetterdaten unsinnige Temperaturwerte herausgefiltert und eine entsprechende Alarmierung ausgelöst werden oder beim Einlesen von Datumswerten falsche Werte zurückgewiesen werden.

Dazu wird ein *Test* in den Programmfluss eingebaut. Je nachdem, ob der Test wahr oder falsch liefert, wird entweder die eine oder die andere Aktion ausgeführt. Anschließend wird wieder in beiden Fällen mit den gleichen Aktionen fortgesetzt.

In C wird dieser Test wie folgt formuliert:

```
if ( test ) dannAnweisung
[ else sonstAnweisung ]
```

Im einfachsten Fall ist unser Test ein Vergleich:

```
ausdruck1 vergleichsOperator ausdruck2
```

Der Ausdruck kann eine Variable, ein fester Wert (Literal) oder auch ein arithmetischer Ausdruck (eine Rechnung) sein. Für den Vergleich können wir folgende Operatoren nutzen:

== Der Test ist wahr, wenn beide Ausdrücke den gleichen Wert haben.

Achtung: Das Gleichheitszeichen muss doppelt geschrieben werden!

!= Der Test ist wahr, wenn die beiden Ausdrücke verschiedene Werte haben.

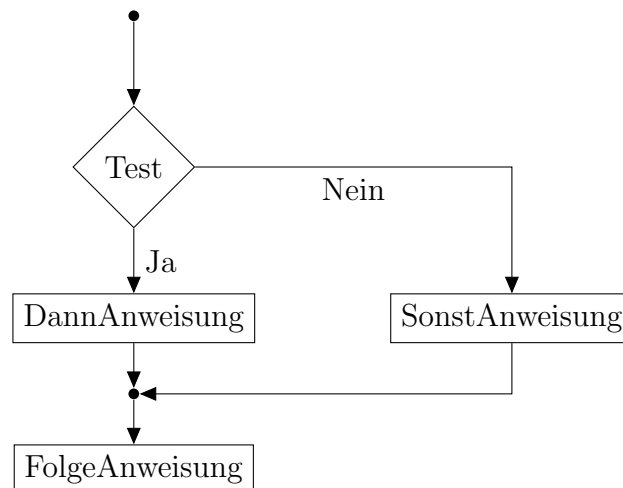


Abb. 3.1: Flussdiagramm einfache Verzweigung

< Der Test ist wahr, wenn der erste Ausdruck kleiner ist als der zweite.

<= Der Test ist wahr, wenn der erste Ausdruck kleiner oder gleich ist zum zweiten.

> Der Test ist wahr, wenn der erste Ausdruck größer ist als der zweite.

>= Der Test ist wahr, wenn der erste Ausdruck größer oder gleich ist zum zweiten.

Warnung: Zum Test auf Gleichheit muss unbedingt das doppelte Gleichheitszeichen == verwendet werden. Ein Test der Form

```

if (a = 3) {
    ...
}
    
```

ist ebenfalls syntaktisch korrekt, bewirkt aber, dass der Variablen `a` der Wert 3 zugewiesen wird. Dieser Ausdruck hat den Wert 3, der als wahr betrachtet wird. Der Test ist also immer erfüllt!

Normalerweise warnt der Compiler in solchen Fällen.

Die Dann- und die Sonst-Anweisung dürfen nur eine einzige Anweisung sein. Wollen wir hier mehr als eine Anweisung unterbringen, müssen wir einen Block bilden, indem wir die gemeinsam auszuführenden Anweisungen in geschweifte Klammern einschließen.

Wir können nun z. B. unsere Funktion zur Flächenberechnung eines Rechtecks so umbauen, dass bei negativer Seitenlänge immer der Wert -1 als Indikator für eine Verletzung des Definitionsbereiches zurückgegeben wird.

Quellcode 3.2: Parametertest bei Funktionsruf

```

int rectArea(int a, int b) {
    if (a < 0) return -1;
    // wird nur erreicht, wenn a >= 0
    if (b < 0) return -1;
    // wird nur erreicht, wenn b >= 0 (und a >= 0)
    return a * b;
}
    
```

Für die Fehlersuche ist es häufig sinnvoll, hier zunächst Ausgaben einzubauen. Wir wollen diese nur im Fehlerfalle und benötigen nun mehr als eine Anweisung in der Dann-Anweisung.

Quellcode 3.3: Parametertest bei Funktionsruf, Ausgabe bei Parameterfehlern

```
int rectArea(int a, int b) {
    if (a < 0) {
        printf("rectArea(), a = %d\n", a);
        return -1;
    }
    // wird nur erreicht, wenn a >= 0
    if (b < 0) {
        printf("rectArea(), b = %d\n", b);
        return -1;
    }
    // wird nur erreicht, wenn b >= 0 (und a >= 0)
    return a * b;
}
```

Bemerkung: Da es bei der Programmentwicklung häufig vorkommt, dass bei Verzweigungen noch Anweisungen (z. B. Testausgaben) eingeschoben werden müssen, bietet es sich an, die Dann- und Sonst-Anweisung generell in geschweifte Klammern einzuschließen, um diese dann später nicht zu vergessen, was zu schwer auffindbaren Fehlern führen kann. Eine Ausnahme sind Verzweigungen, die keinen **else**-Zweig und nur eine kurze Dann-Anweisung besitzen, die wir in die gleiche Zeile wie die **if**-Anweisung schreiben:

```
// kurze Dann-Anweisung:
if (test) dannAnweisung;

// Dann-Anweisung in neuer Zeile:
if (test) {
    dannAnweisung;
    ...
}

// Mit Sonst-Anweisung:
if (test) {
    dannAnweisung;
    ...
} else {
    sonstAnweisung;
    ...
}
```

Als weiteres Beispiel schauen wir uns eine Funktion zum automatischen Testen unseres Algorithmus' an: Wir übergeben dieser Funktion die Parameter der zu testenden Funktion und das erwartete Ergebnis. Diese führt unsere Funktion aus und vergleicht das berechnete mit dem erwarteten Ergebnis. Sind beide Werte gleich, gibt sie 1, sonst 0 zurück (besser wären die logischen Werte *wahr* und *falsch*, die wir später kennenlernen). Eine weitere Funktion `testAreas()` listet die Testfälle auf und zählt die Fehler.

Quellcode 3.4: Funktionstest

```
#include <stdio.h>

/* Flächenberechnung eines Rechtecks.
 * a, b: Seitenlängen
 */
int rectArea(int a, int b) {
```

(Fortsetzung auf der nächsten Seite)

```

    int area = a * b;
    return area;
}

/*
 * Testen von rectArea()
 * a, b: Parameter
 * expected: erwartetes Ergebnis
 * return 0, wenn fehlgeschlagen, sonst 1
 * Ausgabe bei Fehler
 */
int testArea(int a, int b, int expected) {
    int res = rectArea(a, b);
    if (res == expected) {
        // Ausgabe sollte im Erfolgsfall entfallen
        printf("Test (%d, %d) = %d ok\n", a, b, res);
        return 1;
    } else {
        printf("Test (%d, %d) failed: got %d, expected %d\n",
            a, b, res, expected);
        return 0;
    }
}

int testAreas() {
    int errCnt = 0;
    if (testArea(0, 0, 0) == 0) errCnt++;
    if (testArea(1, 1, 1) == 0) errCnt++;
    if (testArea(5, 10, 50) == 0) errCnt++;
    if (testArea(-1, 1, 1) == 0) errCnt++;
    if (testArea(60000, 60000, 36000000000) == 0) errCnt++;
    return errCnt;
}

int main() {
    int errCnt = testAreas();
    if (errCnt > 0) {
        printf("Tests nicht erfolgreich, %d Fehler.\n", errCnt);
    } else {
        printf("Alle Tests ok\n");
    }
}

```

Download: `area1_test.c`.

Aufgabe 3.1 Warum ist der fünfte Test mit (60000, 60000) erfolgreich, obwohl das Ergebnis negativ ist?

Wir können mithilfe der `if`-Anweisung auch testen, ob beim Programmaufruf die korrekte Anzahl der Parameter übergeben wurde. Im Beispiel `geschw1.c` mussten wir dem Programm die Werte für Zeit und Geschwindigkeit übergeben. Fehlen diese, stürzt das Programm ab. Um dies zu vermeiden, prüfen wir in der Funktion `main()` die Anzahl der Parameter und geben einen Fehler aus, wenn diese nicht stimmt. Da die Parameterliste auch den Programmnamen enthält, ist die Anzahl der Parameter um eins größer als die tatsächliche Parameterzahl.

Quellcode 3.5: geschw4.c mit Prüfung der Anzahl der Aufrufparameter

```

/*
 * Streckenberechnung bei gleichförmiger Bewegung.
 */

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

void usage() {
    printf("Aufruf: ./geschw4 zeit geschwindigkeit\n");
    printf("  Zeit in Sekunden\n");
    printf("  Geschwindigkeit in Meter pro Sekunde\n");
}

int main(int argc, char *argv[]) {
    int zeit;           // in Sekunden
    int geschwindigkeit; // in Meter pro Sekunde
    int strecke;        // in Metern

    if (argc != 3) {
        usage();
        return 1;
    }
    zeit = atoi(argv[1]);
    geschwindigkeit = atoi(argv[2]);
    strecke = zeit * geschwindigkeit;
    printf("Zeit %d s, Geschw. %d m/s, Strecke %d m\n",
        zeit, geschwindigkeit, strecke);
}

```

3.5 Zusicherungen

Eine alternative Methode, den Definitionsbereich von Parametern einer Funktion zu prüfen, sind *Assertions* oder Zusicherungen. Wir schließen unseren Test nun nicht in eine Verzweigung ein, sondern übergeben ihn der Funktion `assert()`:

```

#include <assert.h>

...
typ funktionsName(...) {
    assert(test);
    ...
}

```

Wir müssen zusätzlich den Header `assert.h` einbinden, um `assert()` nutzen zu können.

Beispiel: Flächenberechnung mit Zusicherungen. Werden der Funktion nun fehlerhafte Parameter außerhalb des Definitionsbereiches übergeben, wird das Programm mit Angabe der Quelltextzeile, in der die Zusicherung verletzt wurde, abgebrochen.

Quellcode 3.6: Funktion Flächenberechnung mit Zusicherungen

```

#include <stdio.h>
#include <assert.h>

```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```

/* Flächenberechnung eines Rechtecks.
 * a, b: Seitenlängen
 */
int rectArea(int a, int b) {
    assert(a >= 0);
    assert(b >= 0);
    int area = a * b;
    return area;
}

/* Aufruf von rectArea() und Ausgabe des Ergebnisse.
 */
void run(int a, int b) {
    int area = rectArea(a, b);
    printf("Rechteckflaeche bei Seitenlaengen %d und %d ist %d.\n",
        a, b, area);
}

int main() {
    run(3, 5);
    run(3, -5);
    run(-3, 5);
}
    
```

3.6 Rekursive Funktionen

Wir haben bereits gesehen, dass innerhalb einer Funktion andere Funktionen gerufen werden können. Ebenso ist es möglich, dass eine Funktion sich selbst (meist mit veränderten Parametern) ruft, um z. B. Teilergebnisse zu berechnen, die zur Berechnung des Funktionswertes verwendet werden.

Zur Veranschaulichung bietet sich wieder das Modell der kleinen Männchen (kurz Zwerge) an. Der Algorithmus enthält nun Teilaufgaben, die mit dem gleichen Algorithmus, aber neuen Werten ausgeführt werden. Diese übergibt er mit der gleichen Anweisungsliste und die neuen Werte einem weiteren Zwerg und erwartet von ihm das Ergebnis.

Wir wollen z. B. die Summe der Zahlen im Intervall $[a, b]$ berechnen. Dazu müssen wir zur Summe des Intervalls $[a, b - 1]$ lediglich den Wert b addieren. Wir delegieren die Teilaufgabe der Summe von a bis $b-1$ an einen neuen Zwerg. Dieser geht genauso vor, bis ein letzter Zwerg die Summe des Intervalls $[a, a]$ berechnen soll. Er gibt seinem Auftraggeber den Wert a zurück, dieser kann sein Intervall der Länge 2 berechnen, gibt den Wert seinem Auftraggeber zurück u. s. w., bis der erste Auftraggeber die Gesamtsumme ermitteln kann.

Quellcode 3.7: Intervallsumme, rekursiv

```

/* Summe des Intervalls [a, b] rekursiv.
 */
#include <stdio.h>

/* Intervallsumme.
 * a, b: Intervallgrenzen (inklusive)
 * return: Summe
 */
int sumInterval(int a, int b) {
    if (a == b) { // Intervalllänge = 1
        return a;
    }
    return a + sumInterval(a, b - 1);
}
    
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```

    return a;
}
return sumInterval(a, b-1) + b;
}

int main() {
    int a = 3;
    int b = 6;
    int res = sumInterval(a, b);
    printf("summe([%d, %d]) = %d\n", a, b, res);
}

```

Download: `sum_interval.c`.

Der Ablauf nach dem Modell der kleinen Menschen sieht wie folgt aus:

Der erste Zwerg erhält den Algorithmus und die Zahlen 3 und 6.

Er erkennt, dass es mehr als eine Zahl ist und macht weiter.

Er gibt den Algorithmus und die Zahlen 3 und 5 an Zwerg 2.

Der zweite Zwerg erhält die Zahlen 3 und 5.

Er erkennt, dass es mehr als eine Zahl ist und macht weiter.

Er gibt den Algorithmus und die Zahlen 3 und 4 an Zwerg 3.

Der dritte Zwerg erhält die Zahlen 3 und 4.

Er erkennt, dass es mehr als eine Zahl ist und macht weiter.

Er gibt den Algorithmus und die Zahlen 3 und 3 an Zwerg 4.

Der vierte Zwerg erhält die Zahlen 3 und 3.

Er erkennt, dass es nur eine Zahl ist und gibt 3 zurück.

Der dritte Zwerg nimmt die erhaltene 3.

Er addiert 4 und gibt 7 zurück.

Der zweite Zwerg nimmt die erhaltene 7.

Er addiert 5 und gibt 12 zurück.

Der erste Zwerg nimmt die erhaltene 12.

Er addiert 6 und gibt 18 zurück.

Als zweites Beispiel wollen wir die Quersumme einer Zahl im Dezimalsystem berechnen. Mithilfe der Quersumme kann man z. B. feststellen, ob die Zahl durch 3 oder 9 teilbar ist oder diese auch als Prüfziffer verwenden, um sicherzustellen, dass bei der Übertragung keine Zeichenfehler aufgetreten sind.

Um die Quersumme zu bestimmen, müssen wir die einzelnen Ziffern der Zahl im Dezimalsystem bestimmen. Im Computer ist die Zahl jedoch binär gespeichert. Wir benötigen also einen Algorithmus zum Berechnen ihrer Dezimalziffern. Für die *letzte* Ziffer ist das sehr einfach: wir bestimmen einfach den Rest bei Division durch 10:

```
lastDigit = value % 10;
```

Wenn wir diese haben, dann können wir sie abschneiden und können anschließend die vorletzte Ziffer auf die gleiche Weise bestimmen. Auch das Abschneiden einer Ziffer ist einfach: wir teilen die Zahl durch 10

und werfen den Rest weg. In *C* passiert das Beseitigen des Divisionsrestes automatisch, wenn wir ganze Zahlen dividieren:

```
valueWithoutLastDigit = value / 10;
```

Nun müssen wir nur noch das Ende der Berechnung erkennen. Ist unsere Zahl kleiner als 10, dann besteht sie nur aus einer Ziffer und ist damit identisch zu ihrer Quersumme.

Unsere Funktion sieht also wie folgt aus:

```
int digitSum(int val) {
    if (val < 10) return val; // einstellige Zahl == Quersumme
    int lastDigit = val % 10;
    return lastDigit + digitSum(val/10);
}
```

3.7 Aufgaben

Aufgabe 3.2 Erweitern Sie das Programm *area1.c* so, dass Sie die Seitenlängen des Rechtecks als Kommandozeilenparameter übergeben oder die Werte von der Konsole (bzw. einer Datendatei) lesen können.

Aufgabe 3.3 Schreiben Sie ein Programm, das die Funktion *showRectangle()* für verschiedene, vorgegebene Rechtecke aufruft.

Aufgabe 3.4 Erweitern Sie das Rechteckprogramm um weitere Funktionen:

- Berechnung des Umfangs
- Test, ob das Rechteck ein Quadrat ist

Erweitern Sie auch die Ausgabe und schreiben Sie geeignete Tests für diese neuen Funktionen

Aufgabe 3.5 Um eine Temperatur von °C nach °F umzurechnen, benutzen wir die Formel $F = C * 9/5 + 32$. Betrachten Sie folgende beide Implementierungen, die mathematisch identisch sind:

```
int cels2fahr1(int c) {
    int f = c * 9/5 + 32;
    return f;
}

int cels2fahr2(int c) {
    int f = c/5 * 9 + 32;
    return f;
}
```

Schreiben Sie geeignete Tests, um das unterschiedliche Verhalten zu analysieren. Welche der beiden Funktionen ist sinnvoller und warum?

Aufgabe 3.6 Warum funktioniert folgende Funktion für die Fläche eines Quadrats nicht korrekt?

```
int squareArea(int a) {
    if (a < 0)
        printf("squareArea() a=%d\n", a);
    return -1;
    return a * a;
}
```

Aufgabe 3.7 Ziffern einer Zahl

Schreiben Sie Funktionen die

- die Anzahl der Dezimalstellen einer natürlichen Zahl,
- die erste Stelle einer natürlichen Zahl im Dezimalsystem,
- die erste Stelle einer natürlichen Zahl im System zur Basis 8 (Oktalsystem)

bestimmen. Schreiben Sie eine *main()*-Funktion und Tests zum Testen Ihrer Funktionen.

Aufgabe 3.8 Alternierende Quersumme

Schreiben Sie eine Funktion, die die alternierende Quersumme einer Zahl bestimmt: die Ziffern werden dabei abwechselnd addiert und subtrahiert. Beispiel: $\text{altDigitSum}(154) = 1 - 5 + 4 = 0$.

Die Verwendung von `for`, `while` und `goto` ist hier verboten.

Wozu kann die alternierende Quersumme einer Zahl verwendet werden?

- *Datentyp bool*
- *Logische Operatoren*
- *Mehrfachverzweigungen*
- *Aufgaben*

4.1 Datentyp bool

Wir haben bei den Verzweigungen im vorigen Kapitel Testausdrücke benötigt, die entweder wahr oder falsch sind. Ebenso haben wir Testfunktionen geschrieben, deren Rückgabe nur die Werte 1 für *erfolgreich* oder 0 für *nicht erfolgreich* annehmen konnte. Um in Programmen sinnvoll mit solchen Werten *wahr* und *falsch* arbeiten zu können, unterstützen die meisten Programmiersprachen dafür einen eigenen Datentyp. In C wird er seit der Version 99 bereitgestellt. Aus Kompatibilitätsgründen hat dieser den eigenartigen Name `_Bool`, deshalb sollten wir sinnvollerweise das Headerfile `stdbool.h` einbinden, das uns diesen Typ unter dem Namen `bool` und die beiden Literale `true` und `false` bereitstellt.

```
#include <stdbool.h>
```

Bemerkung: Der C-Typ `bool` ist ein ganzzahliger Datentyp, der die Anzahl der erlaubten Werte auf 0 und 1 beschränkt. `false` und `true` sind lediglich andere Bezeichnungen.

Aus diesem Grunde dürfen Testausdrücke auch den Typ `int` (und sogar jeden anderen Datentyp) haben. Der Wert 0 bedeutet stets `false`, jeder andere Wert dagegen wird als `true` interpretiert.

Die Ausgabe von `bool`-Ausdrücken mittels `printf()` erfolgt ebenfalls mit dem Muster für `int`-Werte `%d`, da C die logischen Werte einfach mithilfe von Zahlen darstellt.

Wir können nun unser Testprogramm für die Rechteckfläche etwas verständlicher schreiben, zählen aber nun die erfolgreichen Tests, um die Testausdrücke weiter zu vereinfachen (was praktisch nicht sonderlich sinnvoll ist). Der Testausdruck muss nämlich nicht unbedingt ein Vergleich sein, sondern es kann ein

beliebiger Ausdruck sein, der einen logischen Wert hat. Und unser angepasster Funktionsruf erfüllt diese Bedingung.

Quellcode 4.1: area2_test.c: Tests mit bool-Datentyp

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

/* Flächenberechnung eines Rechtecks.
 * a, b: Seitenlängen
 */
int rectArea(int a, int b) {
    int area = a * b;
    return area;
}

/*
 * Testen von rectArea()
 * a, b: Parameter
 * expected: erwartetes Ergebnis
 * return false, wenn fehlgeschlagen, sonst true
 * Ausgabe bei Fehler
 */
bool testArea(int a, int b, int expected) {
    int res = rectArea(a, b);
    if (res == expected) {
        return true;
    } else {
        printf("Test (%d, %d) failed: got %d, expected %d\n",
            a, b, res, expected);
        return false;
    }
}

/* Test cases
 * Die Tests können weiter vereinfacht werden
 */
int testAreas() {
    int okCnt = 0;
    if (testArea(0, 0, 0)) okCnt++;
    if (testArea(1, 1, 1)) okCnt++;
    if (testArea(5, 10, 50)) okCnt++;
    if (testArea(-1, 1, 1)) okCnt++;
    if (testArea(60000, 60000, 3600000000)) okCnt++;
    return okCnt;
}

int main() {
    int okCnt = testAreas();
    printf("Erfolgreiche Tests: %d\n", okCnt);
}
```

4.2 Logische Operatoren

Oft genügt es nicht, einen einzelnen Vergleich durchzuführen, sondern wir müssen die Ergebnisse mehrerer Vergleiche oder allgemeiner mehrerer Bedingungen miteinander verknüpfen. So sind in kritischen Geräten

Sensoren oft mehrfach vorhanden, und es muss eine Mindestanzahl von ihnen korrekte Werte liefern, damit es zu keiner Notfallreaktion kommt.

Zur Verknüpfung logischer Ausdrücke (z. B. zweier Vergleiche) können wir folgende Operatoren verwenden:

&& Der Gesamtausdruck ist wahr, wenn *beide* Teilausdrücke wahr sind.

Beispiel: `x % 4 == 0 && x % 6 == 0` (die Zahl ist gleichzeitig durch 4 und 6 teilbar).

|| Der Gesamtausdruck ist wahr, wenn *mindestens einer* der beiden Teilausdrücke wahr ist.

Beispiel: `x % 2 == 0 || x % 3 == 0` (der Ausdruck ist wahr, wenn die Zahl durch 2 oder 3 oder auch durch 2 und 3 teilbar ist).

Beispiel: `temp < 18 || temp > 30` (die Temperatur ist außerhalb der zulässigen Bürotemperatur: nur ein Teilausdruck kann hier wahr sein).

! Unärer Operator, negiert den folgenden logischen Ausdruck.

Beispiel: `!(x < 10)` (die Zahl ist mindestens zweistellig)

Um z. B. zu testen, ob die Variable `x` eine zweistellige Zahl ist, würden wir als mathematischen Ausdruck formulieren $10 \leq x < 100$. Versuchen wir dieses Konstrukt in C

```
bool is2digits = 10 <= x < 100;
```

werden wir feststellen, dass wir falsche Ergebnisse erhalten: Es wird zuerst der Vergleich `10 <= x` ausgewertet (im weiteren mit `y` bezeichnet), welcher wahr oder falsch ist, was durch die Werte 1 oder 0 dargestellt wird. Anschließend wird der zweite Teilausdruck `y < 100` ausgewertet, welcher nun immer wahr ist.

Wir müssen stattdessen den Test in zwei Einzeltests zerlegen: eine Zahl ist zweistellig, wenn sie *gleichzeitig*

- größer oder gleich 10 und
- kleiner als 100 ist.

Es müssen also beide Bedingungen erfüllt sein, deshalb verwenden wir die **&&**-Verknüpfung:

```
bool is2digits = (10 <= x && x < 100); // Klammern sind nicht notwendig
```

Um umgekehrt zu testen, ob die Zahl nicht zweistellig ist (oder allgemein außerhalb eines bestimmten Intervalls liegt), können wir die **||**-Verknüpfung verwenden: Die Zahl ist

- kleiner als die untere Intervallgrenze, oder
- größer als die obere Intervallgrenze.

```
bool not2digits = (x < 10 || x >= 100); // Klammern sind nicht notwendig
```

Alternativ können wir auch unseren ersten Ausdruck negieren:

```
bool not2digits = !(10 <= x && x < 100);
```

Die Abarbeitung von logischen Ausdrücken erfolgt immer von links nach rechts und wird abgebrochen, sobald das Ergebnis feststeht. Wenn im obigen Beispiel `x = 1` ist, dann ist der Teilausdruck `10 <= x` bereits falsch, der zweite Test `x < 100` wird nicht mehr ausgeführt.

Wir werden diese Eigenschaft noch häufiger nutzen, um vor dem Ausführen einer Operation deren Zulässigkeit zu prüfen. Bei einer Division dürfen wir nicht durch 0 teilen, wir könnten also schreiben:

```
if (denom != 0 && nom % denom == 0) {
    // Bruch ist kürzbar
    // ...
}
```

Ist der Nenner 0, ist der erste logische Ausdruck falsch und damit die ganze UND-Verknüpfung, egal wie der zweite Ausdruck aussieht. Er wird also nicht ausgeführt, was die Nulldivision und damit den Programmabbruch vermeidet.

Als zweites Beispiel wollen wir eine Funktion schreiben, die uns aus der Monatsnummer (1 bis 12) die Länge des Monats in Tagen (zunächst ohne Berücksichtigung von Schaltjahren) liefert. Wir könnten naiv wie folgt vorgehen:

```
int lenOfMonth(int month) {
    if (month == 1) return 31;
    if (month == 2) return 28;
    ...
    if (month == 12) return 31;
    // Fehlerfall
    return -1;
}
```

Die logischen Operationen erlauben uns eine Zusammenfassung mehrerer Tests zu einem Ausdruck:

```
int lenOfMonth(int month) {
    // Fehlerfall:
    if (month < 1 || month > 12) return -1;
    // Ausnahme Februar
    if (month == 2) return 28;
    // kurze Monate
    if (month == 4 || month == 6 || month == 9 || month == 11) {
        return 30;
    } else { // lange Monate bleiben übrig
        return 31;
    }
}
```

Wir können aber auch die Regelmäßigkeit ausnutzen, dass sich kurze und lange Monate außer zwischen Juli und August abwechseln. Bis Juni sind die geraden, ab September die ungeraden Monate die kurzen:

```
int lenOfMonth(int month) {
    // Fehlerfall:
    if (month < 1 || month > 12) return -1;
    // Ausnahme Februar
    if (month == 2) return 28;
    if ((month < 7 && month % 2 == 0) || (month > 8 && month % 2 == 1)) {
        return 30;
    } else {
        return 31;
    }
}
```

Hier treten im Testausdruck die Operatoren `||` und `&&` gemeinsam auf. In solchen Fällen sollten wir unbedingt Klammern setzen, um die Auswertungsreihenfolge klar zu regeln. Lassen wir sie weg, warnt uns normalerweise der Compiler.

Schreiben wir Testfunktionen, sollten sie im Interesse der Übersichtlichkeit (*Clean Code*) im Erfolgsfall `true` liefern. Oft interessiert uns aber ein fehlgeschlagener Test. Hier kann der `-`Operator sinnvoll eingesetzt werden.

Beispiel: Zählen der fehlgeschlagenen Testfälle bei einem automatischen Programmtest. Wir verwenden wieder unser Programm `area2_test.c`. Wir müssen nun die Fälle zählen, in denen `testArea()` den Wert `false` liefert und negieren dazu den Rückgabewert der Testfunktion:

Quellcode 4.2: Zählen der Fehler durch Negation des Testausdrucks

```
int testAreas() {
    int errCnt = 0;
    if (!testArea(0, 0, 0)) errCnt++;
    if (!testArea(1, 1, 1)) errCnt++;
    if (!testArea(5, 10, 50)) errCnt++;
    if (!testArea(-1, 1, 1)) errCnt++;
    if (!testArea(60000, 60000, 3600000000)) errCnt++;
    return errCnt;
}
```

4.3 Mehrfachverzweigungen

Häufig wollen wir nicht nur zwischen 2, sondern mehreren Fällen unterscheiden. Da die meisten Programmiersprachen aber nur binäre Verzweigungen zulassen, müssen wir Mehrfachverzweigungen aus mehreren Binärverzweigungen zusammensetzen.

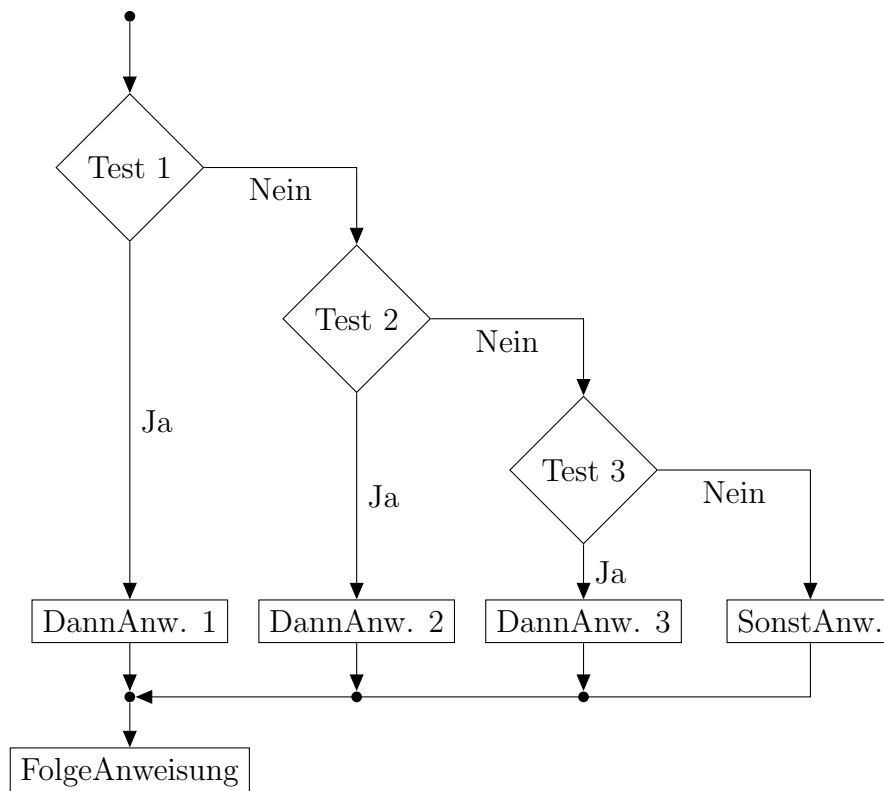


Abb. 4.1: Flussdiagramm Mehrfachverzweigung

Wir testen hier unsere erste Bedingung. Ist diese erfüllt, wird die Anweisung 1 ausgeführt. Ist Sie falsch, testen wir auf die zweite Bedingung. Dafür muss die erste Bedingung nicht noch einmal ausgeschlossen werden, da beim Erreichen von Test 2 nicht erfüllt sein kann. Ist Test 2 erfolgreich, wird die Anweisung für den zweiten Fall ausgeführt, andernfalls wissen wir, dass weder die erste noch die zweite Bedingung erfüllt sein können, und setzen mit dem Test der nächsten Bedingung fort.

In C formulieren wir diese Verzweigung so:

Quellcode 4.3: Mehrfachverzweigung

```

if (test1) {
    dannAnweisung1;
} else {
    if (test2) {
        dannAnweisung2;
    } else {
        if (test3) {
            dannAnweisung3;
        } else {
            sonstAnweisung;
        }
    }
}
folgeAnweisung;

```

Da sich der Code aufgrund der vielen Einrückungen schlecht liest verwenden wir für solche if-else-if-Ketten eine spezielle Formatierung:

Quellcode 4.4: Mehrfachverzweigung mit sinnvollerer Formatierung

```

if (test1) {
    dannAnweisung1;
} else if (test2) {
    dannAnweisung2;
} else if (test3) {
    dannAnweisung3;
} else {
    sonstAnweisung;
}
folgeAnweisung;

```

Bemerkung: In dieser if-else-if-Kette wird immer nur genau eine der Anweisungen abgearbeitet!

Die Zweige sollten unbedingt in geschweifte Klammern eingeschlossen werden, um schwer auffindbare Fehler zu vermeiden.

Beispiel: Berechnung des Steuersatzes (in %) in Abhängigkeit vom Jahreseinkommen (in €) bei einem progressiven Steuersystem. Für jeden Einkommensbereich (< 5000, 5000 ... 9999, ..., 100000) wird ein fester Steuersatz festgelegt.

Quellcode 4.5: taxrate.c Progressiver Steuersatz

```

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

/*
 * Steuersatz in Abhängigkeit vom Einkommen.
 * Die konkreten Werte sind frei erfunden.
 * :param income: Einkommen in EUR
 * :return: Steuersatz in %
 */
int taxRate(int income) {
    int tIncome = income/1000; // in 1000 EUR: Zahlen besser lesbar
    int rate = -1;             // Indikator für Fehler = vergessener Fall

```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```

if (tIncome < 5) {
    rate = 0;
} else if (tIncome < 10) {
    rate = 2;
} else if (tIncome < 50) {
    rate = 5;
} else if (tIncome < 100) {
    rate = 15;
} else { // Einkommen > 100 TEUR
    rate = 25;
}
return rate;
}

void usage() {
    printf("Aufruf: ./taxrate einkommen\n");
}

int main(int argc, char* argv[]) {
    if (argc != 2) {
        usage();
        return 1;
    }
    int income = atoi(argv[1]);
    int rate = taxRate(income);
    printf("Einkommen %d EUR, Steuersatz %d%%\n", income, rate);
}

```

Wenn wir unsere Auswahl durchweg nur anhand konstanter Werte durchführen, können wir in C alternativ die `switch`-Anweisung verwenden:

```

switch (intAusdruck) {
case wert1:
    Anweisung1;
    ...
    break;
case wert2:
    Anweisung2;
    ...
    break;
...
[
else:
    SonstAnweisung;
    ...
    break;
]
}

```

Der Testausdruck wird ausgewertet und von oben nach unten mit den Werten der `case`-Marken verglichen. Sobald der Wert übereinstimmt, werden ab dieser Stelle alle Folgeanweisungen bis zur nächsten `break`-Anweisung abgearbeitet. Beim Erreichen von `break` wird die Abarbeitung nach der `switch`-Anweisung fortgesetzt.

Warnung: Werden die `break`-Anweisungen vergessen, setzt sich die Abarbeitung im nächsten Fall einfach fort, was zu schwer lokalisierbaren Fehlern führen kann.

Aus diesem Grunde ist die Verwendung von `if-else-if`-Ketten vorzuziehen.

Wir wollen nun die `switch`-Anweisung für die Bestimmung der Monatslänge benutzen. Wir könnten diese Funktion wie folgt formulieren:

```
int lenOfMonth(int month) {
    int len; // Rückgabewert Monatslänge

    switch (month) {
    case 1:
        len = 31;
        break;
    case 2:
        len = 28;
        break;
    // ...
    case 12:
        len = 31;
        break;
    default: // Fehlerfall
        len = 0;
        break;
    }
    return len;
}
```

Da viele Monate jeweils die gleiche Länge haben, können wir ausnutzen, dass ohne die `break`-Anweisung die Abarbeitung einfach im nächsten Fall fortgesetzt wird:

```
int lenOfMonth(int month) {
    int len; // Rückgabewert Monatslänge

    switch (month) {
    case 1: // fall through
    case 3: // fall through
    case 5: // fall through
    case 7: // fall through
    case 8: // fall through
    case 10: // fall through
    case 12: // fall through
        len = 31;
        break;
    case 4: // fall through
    case 6: // fall through
    case 9: // fall through
    case 11: // fall through
        len = 30;
        break;
    case 2:
        len = 28;
        break;
    default:
        len = 0;
        break;
    }
}
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```

    return len;
}

```

4.4 Aufgaben

Aufgabe 4.1 *Boolsche Operatoren*

C unterstützt nur die beiden binären Operatoren `&&` und `||`. Wieviele verschiedene binäre Operatoren sind theoretisch möglich? Implementieren Sie fehlende XOR als Funktion und testen Sie die Operation für alle möglichen Parameterpaare.

Aufgabe 4.2 *Schaltjahr*

Schreiben Sie eine Funktion `isLeap()`, die prüft, ob das übergebene Jahr nach dem [Gregorianischen Kalender](#) ein Schaltjahr ist.

Verbesserung: Wenn die Jahreszahl kleiner als 1582 ist, soll die Schaltjahrberechnung nach dem [Julianischen Kalender](#) erfolgen.

Aufgabe 4.3 *Steuersatz*

Erweitern Sie das Programm `taxrate.c` so, dass Sie nicht nur den Steuersatz, sondern auch die tatsächlich zu zahlenden Steuern berechnen (in vollen €).

Aufgabe 4.4 *Binäraddition, Hausaufgabe (5 Punkte)*

Implementieren Sie die Addition zweier dreistelliger Binärzahlen ohne Verwendung der Rechenoperationen `+`, `-`, `*`, `/` und der Bitoperationen. Die logischen Operatoren `&&`, `||` und dürfen nur auf logische Ausdrücke, aber nicht auf die Ziffern selbst angewendet werden. Die Verwendung von Feldern und Schleifen ist hier nicht zulässig.

Die Eingabe der beiden Zahlen erfolgt ziffernweise von der Konsole bzw. per Umlenkung aus einer Datei (genau sechs Ziffern).

Benutzen Sie dazu zwei Funktionen `add()`, die zwei Binärziffern ohne Übertrag addiert (also 0 oder 1 liefert) und `carry()`, die den Übertrag bei der Addition zweier Binärziffern bestimmt. Weitere Funktionen dürfen verwendet werden.

Die Ausgabe soll so erfolgen, dass die beiden Summanden ohne vorlaufende Nullen ausgerichtet übereinander stehen, dann eine Linie und darunter das ausgerichtete Ergebnis folgt.

Beispiel. Eingabe: `booladd1.dat`:

```

0 0 1
0 0 0

```

Erwartetes Ergebnis:

```

  1
  0
----
  1

```


- *Abweisende Schleifen*
- *Verkürzte Zuweisungen*
- *For-Schleifen*
- *Nichtabweisende Schleifen*
- *Schleifensteuerung*
- *Aufgaben*

Wir haben mit *rekursiven Funktionen* bereits eine Möglichkeit kennengelernt, Algorithmen bzw. Anweisungsfolgen wiederholt auszuführen. Viele Programmiersprachen bieten mit den Schleifen eine weitere Möglichkeit, Codeabschnitte wiederholt – meist mit veränderten Daten – auszuführen. Wie auch bei der Rekursion ist es hier erforderlich, diese Wiederholung beim Erreichen eines bestimmten Zustandes zu beenden, was durch einen Test geschieht.

5.1 Abweisende Schleifen

Bei abweisenden Schleifen wird *vor* dem zu wiederholendem Code ein Test ausgeführt, ob eine weitere Wiederholung erforderlich ist. Da dieser Test auch schon beim ersten Aufruf fehlschlagen kann und der Schleifencode dann gar nicht ausgeführt wird, wird diese Schleifenform *abweisende Schleife* genannt. Der Ablauf ist wie folgt:

Nach einer meist erforderlichen Initialisierung wird ein Test ausgeführt. Ist er erfolgreich, dann wird die Schleifenanweisung ausgeführt. Meist ist es erforderlich, die Parameter des Testausdruckes zu modifizieren (z. B. die Zahl der Durchläufe zu erhöhen). Anschließend wird der Test erneut durchgeführt. Solange er erfolgreich ist, wiederholt sich der Vorgang. Schlägt er fehl, dann wird die Abarbeitung nach der Schleife fortgesetzt.

Warnung: Wir müssen sicherstellen, dass der Schleifentest innerhalb der Schleife so modifiziert wird, dass er irgendwann fehlschlägt. Andernfalls gelangen wir in eine *Endlosschleife*, die nie verlassen werden kann.

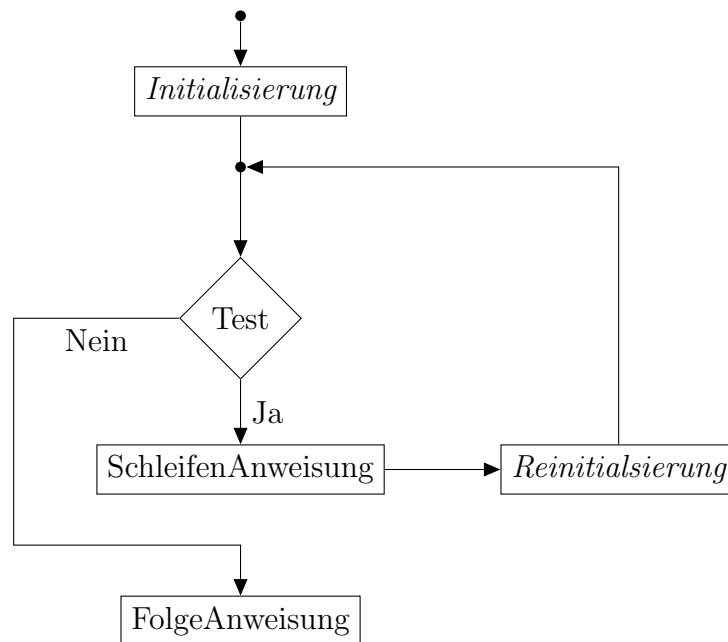


Abb. 5.1: Flussdiagramm einer abweisenden Schleife
Initialisierung und *Reinitialisierung* gehören nicht direkt zur Schleife, werden aber meist benötigt.

Das Programm bleibt hängen und kann nur gewaltsam von außen durch das Betriebssystem beendet werden. Das geschieht mit der Tastenkombination **StrgC**.

Typische Tests sind der Vergleich der Durchläufe mit einer Sollzahl oder ob ein schrittweise zu berechnender Wert nun den korrekten Wert oder die gewünschte Genauigkeit erreicht hat oder ob ein in einer Datenmenge zu suchender Wert gefunden wurde.

Eine abweisende Schleife wird in C mit dem **while**-Konstrukt erzeugt:

```
while (test) schleifenAnweisung;
```

Wir sehen, dass die Syntax innerhalb der Schleife nur eine Anweisung erlaubt. Da wir aber meist mehrere Anweisungen benötigen (Berechnung und Reinitialisierung zur Anpassung des Tests), müssen wir hier im allgemeinen einen Block verwenden.

Im Beispiel wollen wir erneut die Summe der aufeinanderfolgenden Zahlen des Intervalls $[a, b]$ berechnen. Wir initialisieren dazu eine Variable **partSum** mit 0 und addieren sequentiell alle Zahlen von a bis b auf. Ist $b < a$, erhalten wir die Summe 0.

Quellcode 5.1: Intervallsumme

```

/*
 * Summe der Zahlen von a bis b (inklusive)
 */
int sum(int a, int b) {
    int partSum = 0;
    while (a <= b) {
        partSum = partSum + a; // nächsten Wert aufaddieren
        a = a + 1;             // nächster Wert, verändert Testausdruck
    }
}
return partSum;
}
    
```

5.2 Verkürzte Zuweisungen

Da wir gerade im Zusammenhang mit Schleifen häufig den Wert einer Variable um 1 erhöhen oder verringern oder anderweitig durch einen konstanten Wert verändern müssen, bietet C für solche Fälle Kurzschreibweisen:

```
variable op= ausdruck; // bedeutet: variable = variable op (ausdruck)
variable++;           // bedeutet: variable = variable + 1
variable--;           // bedeutet: variable = variable - 1
```

Für `op` können die üblichen binären Operatoren `+`, `-` etc. verwendet werden.

Beispiel: Berechnung einer Potenz a^b . Wir müssen dazu `a` `b`-mal mit sich selbst multiplizieren. Dazu gehen wir analog zur Intervallsumme vor. Wir initialisieren ein Zwischenprodukt mit 1 (warum nicht mit 0?) und multiplizieren es `b`-mal mit dem Wert von `a`.

```
int pow(int a, int b) {
    int prod = 1; // entspricht b^0
    int counter = 0;
    while (counter < b) {
        counter++; // Zahl der Multiplikationen erhöhen
        prod *= a; // Zwischenprodukt mit a multiplizieren
    }
    return prod;
}
```

5.3 For-Schleifen

In abweisenden Schleifen finden wir sehr häufig folgendes Muster:

```
initialisierung;
while (test) {
    schleifenAnweisung;
    reinitialisierung;
}
```

Die Initialisierung und Reinitialisierung sind häufig sehr kurz. Um den Quelltext zu verkürzen, bietet C für solche Fälle eine abgekürzte Schreibweise mit der `for`-Schleife:

`for (initialisierung; test; reinitialisierung) schleifenanweisung;`

Innerhalb der Initialisierung kann eine Variable definiert werden, die nur innerhalb der Schleife verfügbar ist. Eine Schleife, die `n`-mal durchlaufen werden soll, hat meist folgendes Muster:

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

Wir können die Berechnung der Potenz einer Zahl wie folgt vereinfachen:

```
int pow(int a, int b) {
    int prod = 1; // entspricht b^0
    for (int counter = 0; counter < b; counter++) {
        prod *= a; // Zwischenprodukt mit a multiplizieren
    }
    return prod;
}
```

Wollen wir umgekehrt ermitteln, für welchen Exponenten x der Wert von a^x größer oder gleich b ist (diskreter Logarithmus), können wir folgendes Konstrukt verwenden:

Quellcode 5.2: Diskreter Logarithmus

```
int discrLog(int a, int b) {
    int x = 0; // gesuchter Wert
    // x enthält nach Reinitialisierung die Zahl der Multiplikationen
    // Abbruch: pow >= b
    for (pow = 1; pow < b; x++) {
        pow *= a;
    }
    return x;
}
```

Warnung: In vielen Programmiersprache, u. a. in Python, hat die `for`-Schleife eine völlig andere Semantik.

5.4 Nichtabweisende Schleifen

Nichtabweisende Schleifen führen den Schleifenkörper mindestens einmal aus, ehe der Test erfolgt, ob ein weiterer Schleifendurchlauf erforderlich ist.

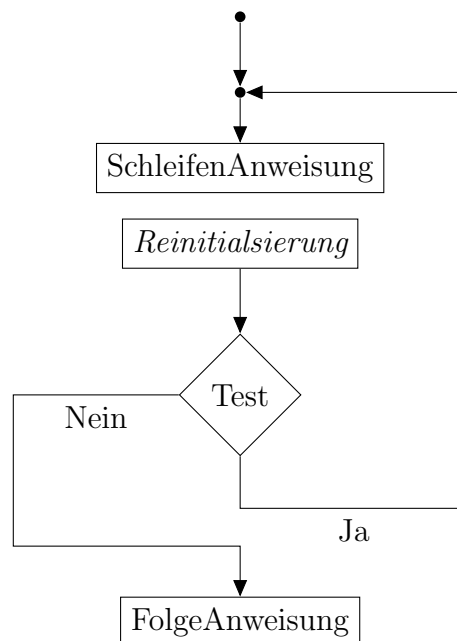


Abb. 5.2: Flussdiagramm einer nichtabweisenden Schleife

In C wird sie mit einer `do-while`-Schleife realisiert:

```
do {
    schleifenanweisung;
    ...
} while (testAusdruck);
```

Die Verwendung eines Blocks ist hier zwingend, auch wenn innerhalb der Schleife nur eine Anweisung benötigt wird. Ist der Test erfolgreich, erfolgt ein erneuter Durchlauf des Schleifenblocks, andernfalls

wird die Schleife beendet.

Beispiel: Einlesen einer Zahlenfolge, bis der Wert -1 auftritt, und Zählen der Werte.

Quellcode 5.3: countseq.c Zählen der Werte einer Sequenz

```
#include <stdio.h>

int readInt() {
    int res;
    scanf("%d", &res);
    return res;
}

/* Einlesen Zahlenfolge, bis Wert -1 auftritt und Zählen
 * der Werte (ohne die -1)
 */
int countSequence() {
    int count = 0;
    int val;
    do {
        val = readInt();
        count++;
    } while (val != -1);
    // Stopzeichen nicht zählen
    return count-1;
}

int main() {
    int cntSeq = countSequence();
    printf("Anzahl Werte %d\n", cntSeq);
}
```

Die nichtweisende Schleife wird relativ selten benötigt und kann einfach durch abweisende Schleifen ersetzt werden, indem man durch geeignete Initialisierung dafür sorgt, dass der Testausdruck vor dem ersten Schleifendurchlauf den Wert `true` liefert.

Quellcode 5.4: Zählen der Werte mit abweisender Schleife

```
/* Einlesen Zahlenfolge, bis Wert -1 auftritt und Zählen
 * der Werte (ohne die -1)
 */
int countSequence() {
    int count = 0;
    int val = 0; // Schleife wird betreten
    while (val != -1) {
        val = readInt();
        count++;
    }
    // Stopzeichen nicht zählen
    return count-1;
}
```

5.5 Schleifensteuerung

Hin und wieder ist es wünschenswert, eine Schleife vorzeitig zu verlassen oder einen Teil des Schleifenkörpers zu überspringen. Beide Aufgaben sind mit den bisher bekannten Sprachkonstrukten problemlos

lösbar. Mit den Schleifensteueranweisungen kann der Code manchmal übersichtlicher gestaltet werden:

break Die Schleifenbearbeitung wird mit dem Erreichen dieser Anweisung sofort abgebrochen, die Arbeit nach der Schleife fortgesetzt.

continue Die folgenden Anweisungen des Schleifenkörpers werden übersprungen, es wird mit der Re-initialisierung der **for**-Schleife bzw. dem Test der **while**-Schleife fortgesetzt.

Beide Anweisungen sind nur sinnvoll in einem Zweig einer Testanweisung. Werden Sie stets ausgeführt, ist die Schleife für **break** überflüssig, da sie maximal einmal bis zu dieser Anweisung ausgeführt werden kann und nie wieder der Schleifentest erreicht wird. Ein **continue** ohne Bedingung macht alle nachfolgenden Anweisungen des Schleifenkörpers überflüssig, da sie nie ausgeführt werden.

Bemerkung: Die Schleifensteueranweisungen sollten nur dann eingesetzt werden, wenn der Code dadurch übersichtlicher wird. Wir sollten versuchen, sie nach Möglichkeit zu vermeiden.

Beispiel: Primzahltest mit **break** und Bestimmen des Mittelwerts der Primzahlen in einem Intervall mit **continue**.

Quellcode 5.5: Primzahltest mit break

```
/* Demonstration von break
 * (kein Clean Code!)
 */
bool isPrime(int p) {
    int d = 2;
    while (d*d <= p) {
        if (p % d == 0) {
            break;
        }
    }
    // Test, ob Schleife abbrochen wurde (d ist Teiler)
    return p % d != 0;
}

/* Mittlere Größe der Primzahlen (gerundet) im Intervall [start, end]
 * Demonstration continue
 * (kein Clean Code!)
 */
int meanPrime(int start, int end) {
    int count = 0;
    int sum = 0;
    for (int p = start; p <= end; p++) {
        if (!isPrime(p)) {
            continue;
        }
        // wird bei erfolgreichem Test übersprungen
        sum += p;
        count++;
    }
    if (count > 0) {
        return sum/count;
    } else {
        return 0;
    }
}
```

5.6 Aufgaben

Aufgabe 5.1 Kann es sinnvoll sein, ein Programm bzw. eine Funktion in einer Endlosschleife laufen zu lassen?

Aufgabe 5.2 Schreiben Sie kleine Programme, die die oben aufgeführten Beispiele testen.

Aufgabe 5.3 Verbessern Sie den Algorithmus für den *diskreten Logarithmus* durch die Verwendung der binären Suche.

Aufgabe 5.4 *Fibonacci-Zahl*

Schreiben Sie eine Funktion zur Berechnung der k-ten Fibonacci-Zahl unter Verwendung *einer* Schleife.

Aufgabe 5.5 *Zahlenfolgen*

Schreiben Sie Funktionen, die jeweils eine Zahlenfolge einlesen, bis der Wert 0 auftritt (dieser Wert gehört nicht zu Folge!) und je einen der folgenden Werte berechnen:

- Minimum der Werte
- Die Größe des Intervalls der Werte (also Maximum - Minimum + 1)
- größte (absolute) Differenz zwischen zwei benachbarten Werten
- größte Abweichung der Werte vom Mittelwert
- Monotonieeigenschaft der Folge. Rückgabewerte:
 - 2: stark monoton wachsend
 - 1: schwach monoton wachsend
 - 0: konstant
 - -1: schwach monoton fallend
 - -2: stark monoton fallend
 - -99: keine Monotonie
- Die größte Summe einer Teilfolge. Beispiel: [-3, 8, -1, 3, 2, -5, 1] liefert 12 = Summe der Teilfolge [8, -1, 3, 2] (*schwierig*)

Aufgabe 5.6 *Collatz-Folge*

Schreiben Sie eine Funktion, die die Länge einer Collatz-Folge $coll(n)$ ermittelt:

- Der erste Wert ist n.
- Ist der k-te Wert der Folge m, so ist der nächste Wert:
 - $m/2$, falls m gerade
 - $3*m+1$, falls m ungerade
- Der letzte Wert der Folge soll 1 sein.

Warum kann man beim Wert 1 sinnvollerweise die Berechnung abbrechen?

Aufgabe 5.7 *Primfaktoren, Hausaufgabe (4 Punkte)*

Schreiben Sie ein Programm, das für jede einer aus einer von der Konsole (oder per Dateiumlenkung aus einer Textdatei) eine Folge positiver ganzer Zahlen einliest, die mit -1 abgeschlossen wird, die Zahl der Primfaktoren ermittelt.

Beispiel:

$$84 = 2 \cdot 2 \cdot 3 \cdot 7, \text{ also } \text{np}(84) = 4$$

Für die Ermittlung der Anzahl der Primfaktoren einzelnen Zahl ist eine eigene Funktion zu schreiben. Weitere Funktionen dürfen verwendet werden.

Beispieleingabe (**np1.dat**):

```
1 2 3 4 6 16 -1
```

Ausgabe für dieses Beispiel:

```
1 0
2 1
3 1
4 2
6 2
16 4
```

Aufgabe 5.8 *Splitfolgen*

Schreiben Sie eine Funktion, die für eine Zahl n folgendes untersucht: Kann ich aus der Folge der Zahlen von 1 bis n zwei Zahlen so herausnehmen, dass ihr Produkt gleich der Summe der restlichen Zahlen ist?

Beispiel: $n = 10$. Wir entnehmen 6 und 7:

$$6 * 7 = 1 + 2 + 3 + 4 + 5 + 8 + 9 + 10 (= 42)$$

Solche Zahlen, für die es derartige Folgen gibt, nennen wir *Splitzahlen*.

Schreiben Sie eine Funktion `countSplit(int n1, int n2)()`, die ermittelt, wieviele solche Zahlen es im abgeschlossenen Intervall $[n_1, n_2]$ gibt. Schreiben Sie ein Testprogramm, dass die Anzahl der Splitzzahlen auf jeweils 100 Zahlen tabellarisch darstellt. Beispiel (Achtung: Werte sind nicht korrekt):

```
1- 100  12
101- 200 13
201- 300  8
...
901-1000 14
```

Aufgabe 5.9 *Kalender*

Schreiben Sie ein Programm, das den Kalender für einen gegebenen Monat in einem bestimmten Jahr tabellarisch anzeigt. Vergleichen Sie Ihr Ergebnis mit dem Unix-Programm **cal** bzw. **ncal**.

Aufgabe 5.10 *Schöne Kuben*

Schreiben Sie eine Funktion, die ermittelt, ob eine Zahl ein *schöner Kubus* ist: die Summe der Kuben (dritten Potenzen) der Ziffern dieser Zahl ergibt die Zahl selbst, Beispiel:

$$407 = 4^3 + 0^3 + 7^3$$

Schreiben Sie ein Testprogramm, das alle schönen Kuben von 1 bis 1000 ermittelt.

Aufgabe 5.11 *Spiegeln einer Zahl*

Schreiben Sie eine Funktion, die eine Zahl im Dezimalsystem spiegelt, z. B. soll `mirror(123)` die Zahl 321 berechnen, `mirror(100)` liefert einfach 1.

Inhalt

- *Datentyp und Fließpunktliterale*
- *Ausdrücke*
- *Automatische Typkonvertierung und Casts*
- *Ein- und Ausgabe*
- *Mathematische Standardbibliothek*
- *Probleme*
- *Algorithmen*
- *Aufgaben*

6.1 Datentyp und Fließpunktliterale

Um auch mit gebrochenen Zahlen arbeiten zu können, benötigen wir einen geeigneten Datentyp. C stellt uns für diese Aufgaben die beiden Typen `double` und `float` zur Verfügung. In den derzeit üblichen Implementierungen auf der *Intel*-Plattform benötigen `float`-Werte 4 Byte im Speicher und haben eine Genauigkeit von 7 Stellen. Der Type `double` benötigt 8 Byte und hat eine Genauigkeit von ca. 17 Stellen. Die Zahl wird intern im Dualsystem in der Form

$$0, \text{mantisse} \cdot 2^{\text{exponent}}$$

dargestellt, so dass nur signifikante Ziffern gespeichert werden müssen. Aufgrund des begrenzten Computerspeichers ist eine exakte Darstellung von gebrochenen und reellen Zahlen prinzipbedingt i. a. nicht möglich, wir müssen immer mit gerundeten Werten arbeiten.

Wenn es keine zwingenden Gründe wie begrenzten Speicherplatz oder die Nutzung von Coprozessoren (Signalprozessoren, *GPUs*) gibt, ist die Nutzung von `double` vorzuziehen.

Eine `double`-Variable wird wie folgt vereinbart:

```
double bezeichner [ = wert ] ;
```

Um Variablen bestimmte Fließpunktwerte zuweisen zu können, müssen wir feste Werte (Literele) im Quellcode darstellen können. Wir schreiben dazu zwischen ganzzahligem Anteil und den Nachkommastellen einen Dezimal**punkt** (ein Komma funktioniert nicht). Bei negativen Werten stellen wir ein Minuszeichen (ohne Leerzeichen) voran.

Für sehr kleine und sehr große Zahlen können wir die wissenschaftliche Darstellung verwenden, die wir auch vom Taschenrechner kennen:

mantisseexponent

Diese Zahl lesen wir als $\text{mantisse} \cdot 10^{\text{exponent}}$. Der Exponent selbst darf ebenfalls ein negatives Vorzeichen haben. Beispiele:

```
3.1415926535
0.0           // Der Dezimalpunkt sichert den Typ double
-7.8
2.99e8        // Lichtgeschwindigkeit in m/s
110e-12       // Größe Silizium-Atom in m (110 Picometer)
```

6.2 Ausdrücke

Ausdrücke mit **double**-Werten werden genauso wie für **int**-Werte erstellt. Nur für die Division gibt es einige Abweichungen:

/ Das Ergebnis ist nun vom Typ **double**.

% Diese Operation steht nicht zur Verfügung.

Beispiel:

```
double a = 3;    // automatische Typkonvertierung des Literals
double b = 5.0;  // Literal hat Typ double
double x;

x = a/b;         // Ergebnis: 0.2
```

Rechnen wir mit Fließpunktzahlen, liefern undefinierte Rechnungen (z. B. Nulldivision) keinen Absturz, sondern einen der Spezialwerte $-\infty$, ∞ oder NaN (not a number).

Beispiel:

```
#include <stdio.h>

int main() {
    printf(" 1/0: %5f\n", 1.0/0.0);
    printf("-9/0: %5f\n", -9.0/0.0);
    printf(" 0/0: %5f\n", 0.0/0.0);
}
```

Ergebnis:

```
1/0:   inf
-9/0:  -inf
0/0:  -nan
```

Diese Werte können nicht einfach durch einen Vergleich geprüft werden, wir benötigen dazu Funktionen aus der *Mathematikbibliothek*.

6.3 Automatische Typkonvertierung und Casts

Treffen innerhalb eines arithmetischen Ausdrucks Werte und Teilausdrücke vom Typ `int` und `double` aufeinander, dann erfolgt eine automatische Konvertierung in den komplexeren Typ `double`, ehe die Operation ausgeführt wird.

Ist bei einer Zuweisung die linke Variable vom Typ `double` und der Ausdruck auf der rechten Seite vom Typ `int`, dann wird das Ergebnis nach `double` umgewandelt. Ist umgekehrt die linke Seite vom Typ `int` und die rechte Seite `double`, dann erfolgt eine Umwandlung nach `int` durch Abschneiden der Nachkommastellen.

Warnung: Bei der Zuweisung von `double`-Werten an `int`-Variablen erhalten wir vom Compiler meist keine Warnung über die erfolgte Rundung.

Bei Zuweisungen von ganzzahligen Ausdrücken an Fließpunktvariablen ist zu beachten, dass der Ausdruck erst *nach* der Berechnung konvertiert wird. Bei folgendem Codeabschnitt erhalten wir nicht das vermutlich gewünschte Ergebnis 0.2, sondern 0:

```
int a = 3;
int b = 15;
double x;

x = a/b; // a/b = 3/15 → 0
```

Um hier die korrekte Fließpunktrechnung zu erzwingen, müssen wir mindestens einen Operanden im Ausdruck nach `double` umwandeln. Das kann z. B. wie folgt geschehen:

```
x = (1.0 * a)/b; // Klammer kann entfallen
```

Durch die Multiplikation mit dem Fließpunktliteral ändert sich mathematisch nichts, aber das Ergebnis wird nach `double` umgewandelt, wodurch auch die Division mit `double`-Werten erfolgt: `b` wird automatisch umgewandelt, da der erste Operand nun eine Fließpunktzahl ist.

Eleganter und besser lesbar ist die Verwendung eines *Casts* (erzwungene Typumwandlung:

`(zieltyp)ausdruck`

Ist der Ausdruck keine Variable, muss er geklammert werden. Wir können also sinnvoller schreiben:

```
x = (double)a/b;
```

Warnung: Der Cast

```
x = (double)(a/b);
```

funktioniert nicht wie gewünscht, da hier die Division wieder ganzzahlig erfolgt und erst danach die Typumwandlung ausgeführt wird.

6.4 Ein- und Ausgabe

Für die Ausgabe von `double`-Werten benutzen wir im Muster der Funktion `printf()` den Platzhalter `%f`. Dieser kann um eine Längenangabe und nach einem Dezimalpunkt um die Anzahl der gewünschten Dezimalstellen erweitert werden.

`%gesamtlänge.nachkommastellenf`

Beispiele:

```
#include <stdio.h>

int main() {
    double PI = 3.14159265358978323;

    printf("%-10s: [%f]\n", "%f", PI);
    printf("%-10s: [%10f]\n", "%10f", PI);
    printf("%-10s: [%8.2f]\n", "%8.2f", PI);
    printf("%-10s: [%20.15f]\n", "%20.15f", PI);
    printf("%-10s: [%.25f]\n", "%.25f", PI);
    printf("%-10s: [%5.0f]\n", "%5.0f", PI);
}
```

Ergebnis:

```
%f      : [3.141593]
%10f    : [  3.141593]
%8.2f   : [    3.14]
%20.15f : [  3.141592653589783]
%.25f   : [3.1415926535897833460353468]
%5.0f   : [    3]
```

Neben der Ausgabeformatierung mit `%f` gibt es noch die Spezifikationen `%e`, die die wissenschaftliche Exponentenschreibweise erzwingt, und `%g`, die automatisch zwischen Darstellung ohne und mit Exponenten auswählt.

Für die Eingabe stellen wir analog zu `readInt()` die Funktion `readDouble()` zur Verfügung:

```
#include <stdio.h>

double readDouble() {
    double res;
    scanf("%lf", &res);
    return res;
}
```

Download `readDouble.inc`.

Wir kopieren dieses Codestück wieder einfach an den Anfang unseres Quelltextes.

Beispiel: Berechnung des Polynoms

$$a_n x^n + a_{n-1} x^{n-1} + \dots + a_1 x + a_0$$

Wir lesen die Werte n , x , a_n , $\dots$, a_0 der Reihe nach von der Konsole (oder per Umlenkung aus einer Datei) ein.

Quellcode 6.1: `polynom.c`: Polynomwertberechnung

```
#include <stdio.h>

/* Polynomwertberechnung
 * Eingabe: n x a_n ... a_0
 */

/* Einlesen von n
 */
int readInt() {
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```

    int res;
    scanf("%d", &res);
    return res;
}

double readDouble() {
    double res;
    scanf("%lf", &res);
    return res;
}

/*  $x^n$ 
 */
double power(double x, double n) {
    double res = 1.0;
    for (int i = 1; i <= n; i++) {
        res *= x;
    }
    return res;
}

/* Wert des Polynoms  $p_n(x)$ .
 * Koeffizienten werden hier gelesen.
 */
double polynomVal(int deg, double x) {
    double res = 0.0;
    for (int n = deg; n >= 0; n--) {
        double a = readDouble();
        res += a * power(x, n);
    }
    return res;
}

int main() {
    int deg = readInt();
    double x = readDouble();
    double val = polynomVal(deg, x);
    printf("p_%d(%f) = %f\n", deg, x, val);
}

```

Bemerkung: Polynomwerte können effizienter mit dem *Horner*-Schema bestimmt werden.

Wollen wir `double`-Werte von der Kommandozeile lesen, benötigen wir die Konvertierungsfunktion `atof`, die wieder durch Einbinden des Headerfiles `stdlib.h` verwenden können.

Beispiel: Lösen der quadratischen Gleichung $ax^2 + bx + c = 0$. Für die Berechnung der Wurzel mit der Funktion `sqrt()` binden wir das Headerfile `math.h` ein und geben beim Übersetzen zusätzlich den Compilerschalter `-lm` an.

Quellcode 6.2: `quadeq.c`: Lösen quadratischer Gleichungen

```

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

```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```

/*
 * Lösen einer quadratischen Gleichung  $ax^2 + bx + c = 0$ 
 * Aufruf: quadeq a b c
 */

/* Lösen der linearen Gleichung  $a*x + b = 0$ 
 */
int solvelin(double a, double b) {
    if (a == 0) {
        if (b == 0) {
            printf("x = 0\n");
            return 1;
        } else {
            printf("Keine Lösung\n");
            return 0;
        }
    }
    double x = - b/a;
    printf("x = %f\n", x);
    return 1;
}

/* Lösen der Gleichung.
 * Lösungen werden ausgegeben (da nur 1 Rückgabewert möglich).
 * :param a, b, c: Koeffizienten
 * :return: Anzahl Lösungen
 */
int solveeq(double a, double b, double c) {
    // Sonderfall a = 0 (lin. Gleichung)
    if (a == 0.0) {
        return solvelin(b, c);
    }
    // Diskriminante
    double d = b*b - 4*a*c;
    if (d < 0) {
        printf("Keine Lösung\n");
        return 0;
    }
    // Erzeugen Normalform  $x^2 + px + q = 0$ 
    double p = b/a;
    double q = c/a;
    // Achtung: Rundungsfehler möglich
    if (d == 0) {
        double x = - p/2;
        printf("x = %f\n", x);
        printf("Probe: %f\n", a*x*x + b*x + c);
        return 1;
    } else {
        double sq = sqrt(p*p/4 - q);
        double x1 = -p/2 + sq;
        double x2 = -p/2 - sq;
        printf("x1 = %f x2= %f\n", x1, x2);
        printf("Probe: %f %f\n", a*x1*x1 + b*x1 + c, a*x2*x2 + b*x2 + c);
        return 2;
    }
}

```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```

}

void usage() {
    printf("Aufruf: sqeq a b c\n");
}

int main(int argc, char* argv[]) {
    if (argc != 4) {
        usage();
        return 1;
    }
    double a = atof(argv[1]);
    double b = atof(argv[2]);
    double c = atof(argv[3]);
    printf("Gleichung: %f x^2 + %f x + %f\n", a, b, c);
    int cnt = solveeq(a, b, c);
    printf("Zahl der Lösungen: %d\n", cnt);
}

```

6.5 Mathematische Standardbibliothek

Für wissenschaftliche Berechnungen benötigen wir sehr häufig mathematische Standardfunktionen, die sich nicht direkt mit den eingebauten Grundrechenarten berechnen lassen. Um diese einfach nutzen zu können, stellt C eine Bibliothek mit den wichtigsten dieser Funktionen zur Verfügung. Um diese nutzen zu können, binden wir in unseren Quellcode das Headerfile `math.h` ein:

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

Damit kennt der Compiler die Signaturen (Aufrufparameter und Rückgabetypen) der benötigten Funktionen. Deren Definition befindet sich aber nicht in der automatisch eingebundenen Standardbibliothek, deshalb müssen wir beim Übersetzen mit dem Schalter `-lm` diese einbinden:

```
gcc -Wall -o programm quelltext.c ... -lm
```

Unter anderem stehen uns folgende Konstanten und Funktionen zur Verfügung:

M_PI $\pi = 3.14159265358979323846$

M_E Eulerkonstante $e = 2.7182818284590452354$

fabs() Betrag der Zahl

floor() Abrunden der Zahl (nächstkleinere ganze Zahl). Das Ergebnis ist wieder vom Typ `double`.

ceil() Aufrunden der Zahl (nächstgrößere ganze Zahl). Das Ergebnis ist wieder vom Typ `double`.

round() Kaufmännisches Runden der Zahl. Das Ergebnis ist wieder vom Typ `double`. (Ab C99 verfügbar)

sqrt() Quadratwurzel der Zahl

exp() Exponentialfunktion der Zahl: e^x

log() Natürlicher Logarithmus der Zahl: $\ln(x)$

sin() Sinus der Zahl (des Winkels). Der Winkel muss in Radiant angegeben werden ($180^\circ = \pi$).

cos() Cosinus der Zahl (des Winkels). Der Winkel muss in Radiant angegeben werden ($180^\circ = \pi$).

asin() Arcusinus der Zahl (Umkehrfunktion des Sinus).

acos() Arcucosinus der Zahl (Umkehrfunktion des Cosinus).

isnan() Liefert einen `int`-Wert ungleich 0 (`true`), wenn das `double`-Argument keine gültige Zahl beschreibt.

isinf() Liefert 1 wenn das Argument $+\infty$, -1, wenn sie $-\infty$ und 0, wenn sie endlich ist.

Da die Winkelfunktionen unseren Winkel nicht in Grad, sondern in Radiant erwarten, schreiben wir uns am besten zwei Konvertierungsfunktionen (die in Python bereits zur Verfügung stehen):

```
/* Konvertiert einen Winkel von Grad nach Radiant
 */
double radians(x) {
    return x * M_PI/180.0;
}

/* Konvertiert einen Winkel von Radiant nach Grad
 */
double degrees(x) {
    return x * 180.0/M_PI;
}
```

6.6 Probleme

Die Verwendung des Datentyps `double` kann bei unbedachter Nutzung zu unerwartetem Verhalten führen. Wir wollen dazu 10-mal den Wert $1/10$ addieren. Als Ergebnis erwarten wir 1.

```
#include <stdio.h>

/* Probleme mit double: Addition von 10-mal 1/10 != 1. */
int main() {
    double x = 0.1;
    double sum = 0.0;
    for (int i = 0; i < 10; i++) {
        sum += x;
    }
    printf("Summe: %f\n", sum);
    if (sum == 1.0) {
        printf("Der Wert ist 1\n");
    } else {
        printf("Der Wert ist ungleich 1\n");
    }
}
```

Führen wir das Programm aus, erhalten wir:

```
Summe: 0.99999999999999988898
Der Wert ist ungleich 1
```

Obwohl der Wert 1 angezeigt wird, scheint er doch ungleich 1 zu sein. Das Problem sind hier minimale Rundungsfehler. Der Wert $1/10$ kann im Dualsystem nicht exakt dargestellt werden, sondern ergibt eine sich wiederholende Periode in den Nachkommastellen (0,0001100110011...). Da der Computer nur eine begrenzte Stellenzahl speichern kann, schneidet er die Nachkommastellen irgendwo ab und macht einen kleinen Fehler. Die Summe ist damit auch ein klein wenig zu klein, so dass wir dies bei der Anzeige der Zahl nicht bemerken, da hier gerundet wird. Der Test prüft jedoch die exakte Gleichheit, die nicht gegeben ist.

Bemerkung: Beim Rechnen mit Fließpunktzahlen dürfen wir bei Vergleichen nie den exakten Wert

prüfen, sondern müssen einen gewissen Rundungsfehler berücksichtigen.

Im Beispiel könnte man z. B. prüfen, ob Ergebnis und erwarteter Wert um weniger als 1 Millionstel ($1e-6$) voneinander abweichen. Sinnvoll ist es allerdings, mit einem relativen Fehler bzgl. der Absolutgröße des erwarteten Werts zu arbeiten.

```
#include <stdio.h>

/* Lösung des Rundungsproblems durch erlaubte Toleranz */

/*
 * Relative Abweichung zwischen gegebenem und erwartetem Wert.
 * diff = |(given-expected)/expected|
 * Falls expected == 0, dann absoluter Fehler
 */

double relDiff(double given, double expected) {
    double rd = given - expected;
    if (expected != 0) {
        rd /= expected;
    }
    // Vorzeichen
    if (rd < 0) rd = -rd;
    return rd;
}

int main() {
    const double REL_ERR = 1e-6;
    double x = 0.1;
    double sum = 0.0;
    for (int i = 0; i < 10; i++) {
        sum += x;
    }
    printf("Summe: %f\n", sum);
    if (relDiff(sum, 1.0) < REL_ERR) {
        printf("Der Wert ist 1 bei relativem Fehler %f\n", REL_ERR);
    } else {
        printf("Der Wert ist ungleich 1\n");
    }
}
```

Wir erhalten nun:

```
Summe: 1.000000
Der Wert ist 1 bei relativem Fehler 0.000001
```

Bei komplizierten Algorithmen können sich die Rundungsfehler sehr stark aufsummieren, so dass die Lösung keinerlei Aussagekraft mehr besitzt. Es gilt also, geeignete Algorithmen zu entwerfen, bei denen dies verhindert wird.

6.7 Algorithmen

Mit diesen Erkenntnissen wollen wir uns nun einige Algorithmen zum Berechnen bekannter Funktionen anschauen. Da diese Funktionen bereits in der C-Mathematikbibliothek zur Verfügung stehen, können wir einfach das Ergebnis vergleichen.

Viele Funktionen können in einer *Taylorreihe* entwickelt werden:

$$f(x) = f(0) + \frac{f'(0)}{1!} \cdot x + \frac{f''(0)}{2!} \cdot x^2 + \frac{f'''(x)}{3!} \cdot x^3 \dots$$

Für den natürlichen Logarithmus $\sin(x)$ erhalten wir durch Berechnen der Ableitungen und Vereinfachen die Formel:

$$\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} \pm \dots$$

Voraussetzung ist, dass die Terme dabei immer kleiner werden, um auf diese Weise Funktionen näherungsweise berechnen zu können. Da die Fakultätsfunktion viel schneller als die Potenzfunktion wächst, ist das hier der Fall.

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h> // zum Vergleich

/* Betrag einer Zahl.
 * in mathlib als fabs() verfügbar
 */
double dabs(double x) {
    if (x >= 0) {
        return x;
    } else {
        return -x;
    }
}

/* Sinus von x mittels Taylorreihe.
 */
double mySin(double x, double err) {
    double sum = x;
    int sign = -1; // zweiter Term negativ
    double term = 1 + err; // nächster Term, Dummywert > err für Schleifentest
    double xpot = x; // x^1
    double denom = 1; // 1!
    for (int pot = 3; dabs(term) > err; pot +=2) {
        xpot *= x; // Potenz um 2 erhöhen
        denom *= (pot-1)*(pot); // Fakultät um 2 weiter
        term = sign*xpot/denom;
        sum += term;
        sign = -sign; // Vorzeichen wechseln
    }
    return sum;
}

int main(int argc, char *argv[]) {
    double x = atof(argv[1]);
    const double ERR = 1e-6;
    printf("sin(%f): Taylor=%f Build-in=%f\n",
        x, mySin(x, ERR), sin(x));
}
```

Für die Wurzelfunktion können wir nicht so vorgehen, da die Taylorreihe nur für Werte $0 \leq x \leq 2$ konvergiert. Deshalb gehen wir hier anders vor und verwenden das *Newton*-Verfahren. Dazu stellen wir die Gleichung $x = \sqrt{a}$ in ein Polynom um, von dem wir die Nullstelle suchen:

$$f(x) = x^2 - a = 0$$

Ist diese Funktion differenzierbar und ihre Ableitung stetig, können wir folgenden Algorithmus (*Newton-Verfahren*) verwenden:

$$x_0 = a$$

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}, n = 1, 2, \dots$$

Setzen wir unsere konkrete Funktion ein:

$$f(x) = x^2 - a$$

$$f'(x) = 2 \cdot x, \text{ und damit}$$

$$x_{n+1} = x_n - \frac{x_n^2 - a}{2x_n}$$

$$= \frac{1}{2} \left(\frac{a}{x_n} + x_n \right)$$

Wir führen die Berechnung solange durch, bis sich der Wert von x_n kaum noch ändert, indem wir die relative Differenz

$$\varepsilon = \left| \frac{x_{n+1} - x_n}{x_{n+1}} \right|$$

berechnen und abbrechen, wenn diese kleiner als die benötigte Genauigkeit ist.

Quellcode 6.3: newton.c: Wurzelberechnung nach dem Newton-Verfahren

```
/* Wurzelberechnung mit dem Newton-Verfahren.
*/

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

/* Absolutbetrag.
*/
double dabs(double x) {
    if (x >= 0) {
        return x;
    } else {
        return -x;
    }
}

/* Wurzelberechnung nach Newton-Verfahren.
 * :param a: Parameter
 * :param err: maximaler Relativer Fehler
 * :return: Wurzel von a
 */
double newtonRoot(double a, double err) {
    if (a < 0) return -1;
    double xNew = a; // Startwert x(0)
    double xOld = 0; // Dummywert, klappt für err < 1
    while (dabs(xNew - xOld)/xNew > err) {
        xOld = xNew;
        xNew = (a/xOld + xOld)/2.0;
    }
    return xNew;
}
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```

void usage() {
    printf("Usage: ./newton value\n");
}

int main(int argc, const char * argv[]) {
    if (argc != 2) {
        usage();
        return 1;
    }
    const double ERR = 1e-6;
    double a = atof(argv[1]);
    double res = newtonRoot(a, ERR);
    printf("sqrt(%f) = %f\n", a, res);
    printf("Quadratischer Fehler: %e\n", fabs(res * res - a));
}

```

Bemerkung: Dieser konkrete Algorithmus für die Berechnung der Quadratwurzel war schon den Babyloniern vor ca. 4000 Jahren bekannt und wurde vor ca. 2000 Jahren von *Heron* veröffentlicht, deshalb kennen wir es auch als *Heron-Verfahren*. Das *Newton-Verfahren* ist der allgemeinere Ansatz für die Nullstellenberechnung beliebiger differenzierbarer Funktionen.

6.8 Aufgaben

Aufgabe 6.1 Warum erhalten wir eine Warnung bei der Definition

```
double pi = 3,141;
```

Welchen Wert enthält `pi`?

Aufgabe 6.2 *Berechnung von*

Schreiben Sie eine Funktion zur Berechnung von π : Gehen Sie von einem regelmäßigen Sechseck aus, das in einem Kreis vom Radius 1 eingeschlossen ist. Es hat den Umfang 6, der Kreis dagegen den Umfang 2π . Verdoppeln Sie nun die Eckenzahl und berechnen den Umfang des 12-Ecks. Wiederholen Sie die Verdoppelung, bis sich der Umfang des n -Ecks kaum noch ändert.

Aufgabe 6.3 *Kubikwurzel*

Berechnen Sie die dritte Wurzel einer Zahl nach dem Newton-Verfahren. (Diese Funktion gibt es übrigens nicht in der C-Mathematikbibliothek).

Aufgabe 6.4 *Probleme Newton-Verfahren*

Schreiben Sie ein Programm, das die Nullstelle von $x^3 - 2x + 2$ mit dem Newtonverfahren ermittelt. Was passiert, wenn Sie als Startwert 0 verwenden? Was passiert bei anderen Startwerten?

- *Motivation*
- *Aufbau, Vereinbarung und Zugriff*
- *Exkurs: Graphische Präsentation der Ergebnisse*
- *Mehrdimensionale Felder*
- *Felder und Funktionen*
- *Einfügen und Löschen von Feldelementen*
- *Aufgaben*

7.1 Motivation

Häufig benötigen wir eine größere Anzahl von Werten des gleichen Typs, z.B. eine Liste von Messwerten, Umsatzzahlen oder Strings. Solange die Zahl sehr klein und von vornherein bekannt ist, können wir mit Einzelvariablen `wert1`, `wert2`, ... arbeiten. Sobald die Zahl aber nicht genau bekannt ist oder größer wird, ist dieses Vorgehen nicht praktikabel. Wir benötigen eine Möglichkeit, einen Algorithmus auf eine Menge gleichartiger Variablen automatisch anwenden zu können. Diese Möglichkeit bieten uns die *Felder* (*Arrays*) in C.

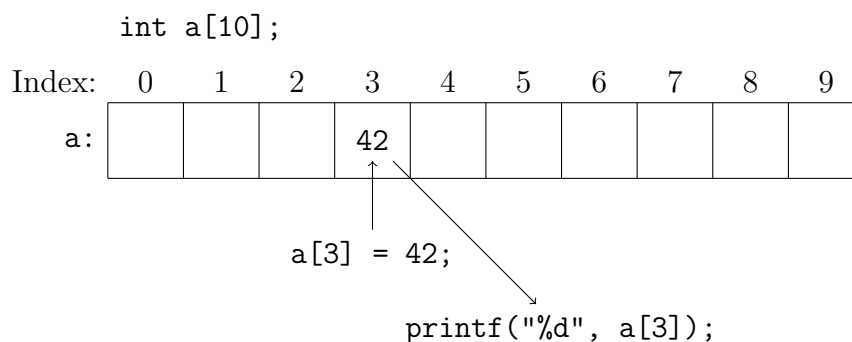
7.2 Aufbau, Vereinbarung und Zugriff

Zum Anlegen eines Feldes benötigen wir den Typ sowie die Anzahl der Feldelemente:

```
typ bezeichner[größe] [= {initialisierungsliste}] ;
```

Es entstehen die gewünschte Anzahl von Feldelementen, die unmittelbar hintereinander im Speicher angelegt werden. Geben wir keine Initialisierung an, enthalten sie wie bei einfachen Variablen unbestimmte Werte.

Eine Initialisierungsliste besteht aus durch Komma getrennten Werten, mit denen die Feldelemente der Reihe nach belegt werden. Ist diese angegeben, dann kann die Größenangabe für das Feld entfallen, sie



wird automatisch von der Länge der Liste abgeleitet. Ist die Initialisierungsliste kürzer als das Feld, werden die restlichen Elemente mit 0 initialisiert. Eine längere Initialisierungsliste als die Feldgröße ist dagegen nicht zulässig.

Beispiele:

```
int a[3];           // int-Feld mit 3 Elementen, uninitialisiert
bool b[5];          // bool-Feld mit 5 Elementen, uninitialisiert
int a[3] = {3, 2, 1}; // int-Feld mit 3 Elementen mit Werten 3, 2, 1
int a[] = {3, 2, 1};  // ditto
int a[3] = {3, 2};    // int-Feld mit 3 Elementen mit Werten 3, 2, 0
int a[3] = {3};       // int-Feld mit 3 Elementen mit Werten 3, 0, 0
int a[2] = {1, 2, 3}; // nicht zulässig
```

Jedes Feldelement verhält sich wie eine einfache Variable vom gleichen Datentyp. Es wird mit dem Konstrukt

`feldName[index]`

angesprochen. Der Index ist ein `int`-Ausdruck, der die Werte von 0 für das erste Feldelement bis zu Größe-1 für das letzte Feldelement annehmen darf.

Warnung: Unzulässige Indexwerte außerhalb der Feldgröße werden nicht geprüft und führen zu unvorhersehbaren Programmfehlern und Sicherheitsrisiken!

Der Index für das erste Feldelement wurde (im Gegensatz zu *Pascal* oder *R*) 0 und nicht 1 gewählt, um den Feldzugriff zu beschleunigen. Die Position eines Feldelements lässt sich so einfacher berechnen:

```
position(i) = startPositionFeld + index * elementBreite
```

Das Muster einer Schleife, die alle Feldelemente durchläuft, hat deshalb meist folgenden Aufbau:

```
for (int i = 0; i < feldGroesse; i++) {
    // bearbeite feld[i]
}
```

Beispiel: Erstellen eines Histogramms der Zensurenverteilung einer Klausur (Zensuren 1 bis 5). Die Zensuren sollen von der Konsole eingelesen werden, die Anzahl ist nicht vorher bekannt, wir schließen die Eingabe mit dem Wert 0 ab.

Quellcode 7.1: zensuren1.c: Zensurenspiegel

```
/*
 * Histogramm der Zensurenverteilung einer Klausur.
 * Eingabe Werte 1 ... 5 über Konsole, Abschluss mit 0.
 */
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```

#include <stdio.h>
#include <assert.h>

int readInt() {
    int res;
    scanf("%d", &res);
    return res;
}

int main() {
    int hist[6] = {0}; // Zensuren 1...5, hist[0] für ungültige Werte
    // Eingabe
    // TODO: eigene Funktion
    int val = readInt();
    int sumZensuren = 0; // für Probe
    while (val != 0) {
        sumZensuren += val;
        if (val < 0 || val > 5) { // ungültig
            hist[0]++;
        } else {
            hist[val]++; // zensur = Feldindex
        }
        val = readInt(); // nächsten Wert lesen
    }
    // Probe: summe (anzahl*zensur) = summe (zensuren)
    int sumHist = 0;
    for (int zensur = 1; zensur <= 5; zensur++) {
        sumHist += zensur * hist[zensur];
    }
    // Ausgabe
    // TODO: eigene Funktion
    if (hist[0] > 0) {
        printf("# ungültig: %d\n", hist[0]);
    }
    printf("#Zensur Anzahl\n");
    for (int i = 1; i <= 5; i++) {
        printf("%7d %2d\n", i, hist[i]);
    }
    // Prüfung Gleichheit beider Summen, sollte nur bei fehlerhaften Eingaben
    // Abbruch erzwingen
    assert (sumZensuren == sumHist);
}

```

7.3 Exkurs: Graphische Präsentation der Ergebnisse

Wollen wir große Zahlenmengen präsentieren, bietet sich häufig eine graphische Darstellung an. Im Praktikum *Statistik mit R* benutzen wir dazu die Programmiersprache *R*, im Kurs *Machine Learning* verwenden wir die *Python*-Bibliothek *Matplotlib*. Ein Kommandozeilenwerkzeug zum automatisierbaren Erstellen von ansprechenden Diagrammen ist *Gnuplot*, das wir uns hier kurz anschauen wollen.

Wir können Gnuplot entweder interaktiv nutzen, indem wir es ohne Parameter starten, oder wir können ihm als Parameter eine Steuerdatei mit Gnuplot-Befehlen mitgeben, um automatisch Diagramme (dann meist in Form von Graphikdateien) zu erzeugen.

Zum interaktiven Start geben wir auf einer Konsole einfach

gnuplot

ein befinden uns nun auf Gnuplot-Kommandozeile. Zum eine Funktion zu plotten, verwenden wir den Befehl

```
plot f(x)
```

wobei f ein Ausdruck für die gewünschte Funktion ist. Der Parameter muss stets x sein. Beispiele:

```
plot 1/(1+x**4)
plot sin(3*x+5)
```

Wir können zusätzlich das gewünschte Intervall für den Parameter und auch den Funktionswert angeben:

```
plot [x0:x1] [y0:y1] f(x)
```

Beispiel:

```
plot [-5:5] abs(x)/(1+x**2)
plot [-2:2] [-1:1] sin(3*x+5)
```

Um den Graph in einer Datei zu speichern, verwenden wir die beiden Befehle

```
set term format
set output "dateiname"
```

Das Format kann z. B. **svg** oder **pdf** für skalierbare oder **png** für Rastergraphiken sein. Für hochwertige Ausgaben sollten immer skalierbare Bildformate zum Einsatz kommen.

Beispiel:

```
set term svg
set output 'plot-ln.svg'
plot [0.5:5] log(x)
```

Ergebnis:

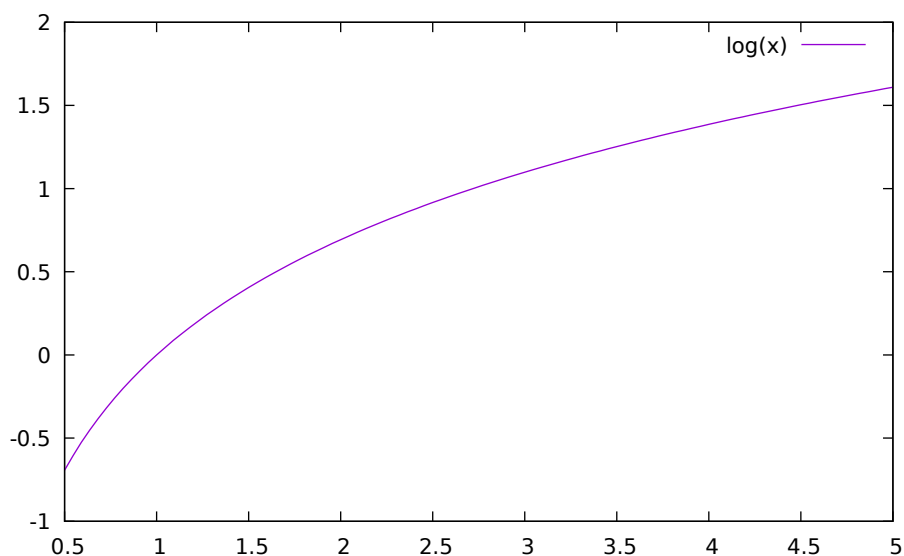


Abb. 7.1: Funktionsgraph von $\ln(x)$

Wichtiger für uns die graphische Darstellung von beobachteten oder berechneten Daten. Wir benötigen dazu im einfachsten Falle eine Textdatei mit zwei Zahlenspalten, die durch Leerzeichen oder Tabulator getrennt sind.

Wir betrachten dazu unseren Zensurenspiegel. Beispiel:

```
#Zensur Anzahl
1 2
2 3
3 4
4 1
5 2
```

Eine solche Tabelle können wir wie folgt darstellen:

```
plot 'dateiname' [with methode]
```

Ohne weitere Angabe werden einfach Symbole (z. B. Kreuzchen) an den entsprechenden x-y-Positionen dargestellt. Statt dessen können wir eine Methode angeben:

lines Verbinden der Punkte durch Linien.

boxes Boxplot, Säulendiagramm.

Sollen die Boxen ausgefüllt werden, fügen wir vorher den Befehl

```
set style fill solid
```

ein. Den Legendeneintrag können wir innerhalb des **plot**-Befehls durch den Parameter **title** oder kurz **ti** ändern:

```
plot ... ti 'legendenTitel'
```

Die Achsenbeschriftung geschieht mit den Befehlen **xset** und **yset** vor dem **plot**-Befehl:

```
set xlab 'xBezeichnung'
set ylab 'yBezeichnung'
```

Beim Barplot sollten noch x- und y-Intervall angegeben werden.

Beispiel Liniengraph und Boxplot des Zensurenspiegels. Der **using**-Parameter verbessert die Achsenbeschriftung und kann entfallen.

```
set xlab 'Zensur'
set ylab 'Anzahl'
plot 'zensuren.out' with lines ti 'Anzahl je Zensur'

set style fill solid
plot [0:6][0:6] 'zensuren.out' using 1:2:xtic(1) with boxes ti 'Anzahl je Zensur'
```

Ergebnis:

Um die interaktive Konsole von Gnuplot zu verlassen, geben wir den Befehl

```
quit
```

ein. Rufen wir dagegen Gnuplot mit einer Steuerdatei auf, dann beendet es sich nach der Abarbeitung automatisch. Um in diesem Falle die erzeugten Diagramme anzuzeigen, fügen wir nach jedem **plot**-Befehl eine Pause ein:

```
pause -1 [ 'anzeigeText' ]
```

7.4 Mehrdimensionale Felder

Oft benötigen wir nicht nur eine Liste von Werten, sondern eine Matrix. So wollen wir bei einer Klausur die Punktzahlen für jeden Teilnehmer für jede Aufgabe einzeln abbilden, oder wir interessieren uns für eine bestimmte Kennzahl (z. B. Zahl der Geburten) pro Jahr für jedes Bundesland. Immer, wenn unsere Daten vom gleichen Typ sind und von mehreren ganzzahligen Parametern (den Indices) abhängig sind, können wir diese sinnvoll als mehrdimensionales Feld abbilden.

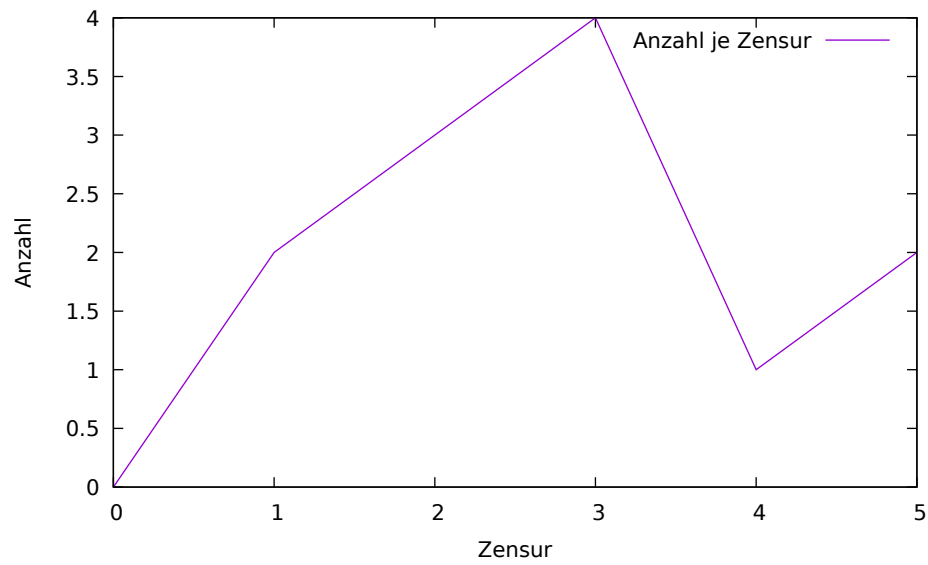


Abb. 7.2: Zensurenspiegel als Liniendiagramm

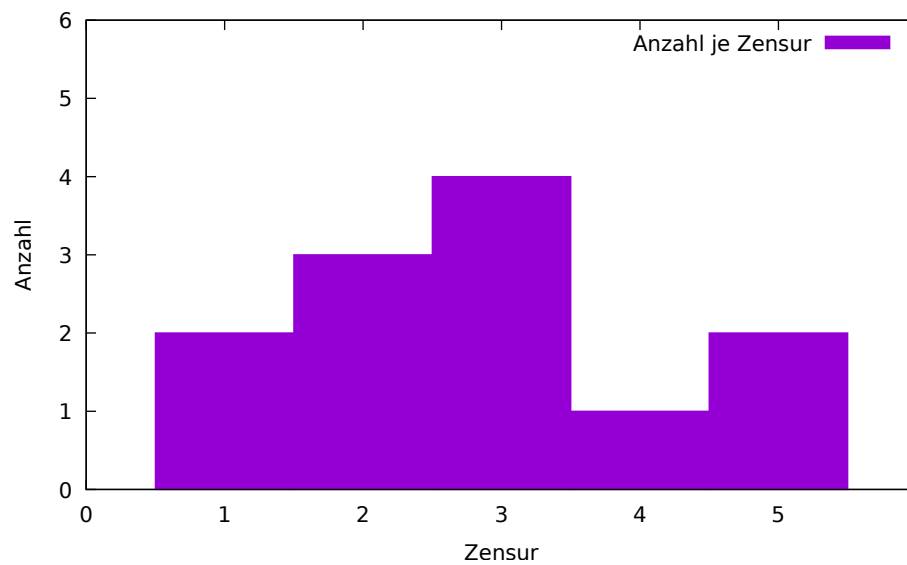


Abb. 7.3: Zensurenspiegel als Boxplot

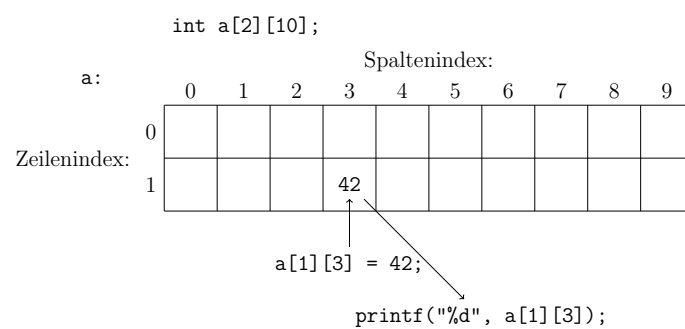


Abb. 7.4: Aufbau zweidimensionales Feld.

Die Vereinbarung einer solchen Feldvariable erfolgt analog zu eindimensionalen Feldern:

```
typ feldVariable[dim0][dim1]... [= {initialisierung}];
```

Die Dimensionen dürfen seit C99 Variablen sein, eine Anfangsinitialisierung ist aber nur dann möglich, wenn die Dimensionen zur Compilierungszeit bekannte Konstanten sind. Wenn eine Initialisierung angegeben ist, dann kann die Angabe der Dimensionen weggelassen werden, sie wird von den Initialisierungswerten abgeleitet.

Die mehrdimensionale Initialisierung erfolgt durch verschachtelte Listen, die in geschweifte Klammern eingeschlossen werden:

```
{
    {werteZeile0},
    ...
    {werteZeile0}
}
```

Die Werte einer Zeile sind wiederum durch Kommata getrennt.

Beispiele:

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

int main(int argc, char *argv[]) {
    // Vereinbarung const int ROWS = 4 funktioniert nicht für a2
    enum {ROWS = 4, COLS = 3};

    // Parameter für variable Größe
    if (argc != 3) {
        printf("Aufruf:  define2d zeilen spalten\n");
        return 1;
    }
    int rows = atoi(argv[1]);
    int cols = atoi(argv[2]);
    // feste Größe
    int a1[ROWS][COLS];

    // feste Größe, initialisiert
    int a2[ROWS][COLS] = {
        {1, 2, 3},
        {4, 5, 6},
        {7, 8, 9},
        {10, 11, 12}
    };

    // feste Größe, erste Dimension aus Initialisierung abgeleitet
    int a3[][COLS] = {
        {1, 2, 3},
        {4, 5, 6},
        {7, 8, 9},
        {10, 11, 12}
    };

    // funktioniert nicht:
    // int a4[][] = { {...}, ...};

    // Größe erst zur Laufzeit bekannt, Initialisierung bei Def. nicht möglich
    int a5[rows][cols];
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
}
```

Zweidimensionale Felder sind intern zeilenweise organisiert, i. e. die Zeilen eines Feldes werden miteinander verkettet und zusammenhängend im Speicher abgelegt.

Der Zugriff auf Elemente mehrdimensionaler Felder erfolgt analog zu eindimensionalen Feldern. Jeder Index wird in ein eigenes Paar eckiger Klammern [] eingeschlossen.

`feldVariable[zeilenIndex][spaltenIndex]...`

Beispiel: Multiplikationstabelle.

Quellcode 7.2: multtable1.c: Multiplikationstabelle

```
/* Multiplikationstabelle, kleines 1x1.
 */

#include <stdio.h>

enum {MAX_FACTOR = 10}; // Größe Tabelle

int main() {
    int mult[MAX_FACTOR+1][MAX_FACTOR+1]; // um 1 vergrößern wg. Index 0
    // Erzeugen
    for (int row = 0; row <= MAX_FACTOR; row++) {
        for (int col = 0; col <= MAX_FACTOR; col++) {
            mult[row][col] = row * col;
        }
    }
    // Ausgabe
    // 1. Spaltenköpfe
    printf("* |");
    for (int col = 0; col <= MAX_FACTOR; col++) {
        printf("%4d", col);
    }
    printf("\n");
    // 2. Trenner
    printf("--|");
    for (int col = 0; col <= MAX_FACTOR; col++) {
        printf("----");
    }
    printf("\n");
    // 3. Tabelle
    for (int row = 0; row <= MAX_FACTOR; row++) {
        // Zeilenkopf
        printf("%2d|", row);
        for (int col = 0; col <= MAX_FACTOR; col++) {
            printf("%4d", mult[row][col]);
        }
        printf("\n");
    }
}
```

Ergebnis:

*		0	1	2	3	4	5	6	7	8	9	10
--		-----										
0		0	0	0	0	0	0	0	0	0	0	0

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

1	0	1	2	3	4	5	6	7	8	9	10
2	0	2	4	6	8	10	12	14	16	18	20
3	0	3	6	9	12	15	18	21	24	27	30
4	0	4	8	12	16	20	24	28	32	36	40
5	0	5	10	15	20	25	30	35	40	45	50
6	0	6	12	18	24	30	36	42	48	54	60
7	0	7	14	21	28	35	42	49	56	63	70
8	0	8	16	24	32	40	48	56	64	72	80
9	0	9	18	27	36	45	54	63	72	81	90
10	0	10	20	30	40	50	60	70	80	90	100

7.5 Felder und Funktionen

Wir können an Funktionen auch Felder als Parameter übergeben, müssen dabei allerdings einige Besonderheiten beachten. Zunächst wird die Angabe der ersten Dimension eines Feldes ignoriert, wir können diese deshalb auch weglassen:

```
rückgabeTyp f1(feld[] ...) ...
rückgabeTyp f2(feld[][ dim2] ...) ...
```

Alle weiteren Dimensionen bei mehrdimensionalen Feldern müssen jedoch angegeben sein. Benötigt unsere Funktion die tatsächliche Feldgröße, da diese zur Compile-Zeit nicht bekannt ist, müssen wir diese als zusätzlichen Parameter übergeben. Beispiel:

```
int listSum(int a[], int len) {
    int sum = 0;
    for (int i = 0; i < len; i++) {
        sum += i;
    }
    return sum;
}
```

Im Gegensatz zu einfachen Variablen und *Structs* werden Felder als *Referenzen* übergeben (genauer genommen wird ein *Pointer* auf das erste Feldelement übergeben). Das bedeutet, dass unsere Funktion bei Änderungen im Feld das Originalfeld bearbeitet.

Beispiel: Änderung einer Preisliste bei neuem Umsatzsteuersatz.

```
// Raten relativ zu 1, keine Prozentangabe!
void changePrices(double prices[], int len, double oldRate, double newRate) {
    for (int i = 0; i < len; i++) {
        prices[i] = prices[i]/(1.0+oldRate)*(1.0+newRate);
    }
}
```

Ein Feld kann von einer Funktion nicht als Rückgabetyt verwendet werden. Folgender Code ist unzulässig:

```
// Funktioniert nicht!
int[] createArray(int len) {
    int a[len];
    return a;
}
```

Der Grund ist, dass das Feld lokal innerhalb der Funktion angelegt und damit beim Beenden der Funktion freigegeben wird. Die Funktion gibt aber eine Referenz auf das erste Feldelement zurück, die nun auf freigegebenen Speicher zeigt.

Benötigen wir als Rückgabewert ein Feld, dann müssen wir das Feld in der aufrufenden Funktion anlegen und dieses Feld als zusätzlichen Parameter mitgeben.

Beispiel: Einlesen einer Folge von maximal 1000 Zahlen in ein Feld. Die Folge kann vorher mit -1 beendet werden.

Quellcode 7.3: readSeq.c: Zahlenfolge in Funktion einlesen, Feld-rückgabe

```

/* Einlesen einer Zahlenfolge bis -1, maximal maxLen Werte.
 */

#include <stdio.h>

int readInt() {
    int res;
    scanf("%d", &res);
    return res;
}

/* Lesen der Folge und Ablage in a.
 * :return: Anzahl der tatsächlich gelesenen Werte.
 */
int readSeq(int a[], int maxLen) {
    int val = readInt();
    int len = 0;
    while (len < maxLen && val != -1) {
        a[len] = val;
        len++;
        val = readInt();
    }
    return len;
}

void showSeq(int a[], int len) {
    for (int i = 0; i < len; i++) {
        printf(" [%d]: %d", i, a[i]);
    }
    printf("\n");
}

int main() {
    int maxLen = 1000;
    int a[maxLen];
    int len = readSeq(a, maxLen);
    // Ausgabe
    showSeq(a, len);
}

```

Beispiel: Zensurenspiegel. Wir erledigen nun das Einlesen, das Auswerten und die Ausgabe des Histogramms in eigenen Funktionen mit dem Feld als Parameter.

Quellcode 7.4: zensuren2.c: Einlesen, Auswerten, Ausgabe als Funktionen

```

/*
 * Histogramm der Zensurenverteilung einer Klausur.
 * Eingabe Werte 1 ... 5 über Konsole, Abschluss mit 0.

```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```

/* Eingabe, Berechnung Durchschnitt, Ausgabe in Funktionen
*/

#include <stdio.h>
#include <assert.h>

const int MAX_GRADE = 5; // Höchste Zensur, ganzzahlig

int readInt() {
    int res;
    scanf("%d", &res);
    return res;
}

/* hist[] auf 0 initialisieren.
*/
void initHist(int hist[]) {
    for (int i = 0; i <= MAX_GRADE; i++) {
        hist[i] = 0;
    }
}

/* Zensuren in Histogramm lesen.
* :return: Anzahl Zensuren
*/
int readHist(int hist[]) {
    initHist(hist);
    int count = 0;
    int val = readInt();
    while (val != 0) {
        count++;
        if (val < 0 || val > MAX_GRADE) { // ungültig
            hist[0]++;
        } else {
            hist[val]++; // zensur = Feldindex
        }
        val = readInt(); // nächsten Wert lesen
    }
    return count;
}

/* Zensurendurchschnitt.
* :return: Durchschnitt
*/
double meanHist(int hist[]) {
    // Ignoriere Fehler
    int count = 0;
    int sum = 0;
    for (int grade = 1; grade <= MAX_GRADE; grade++) {
        count += hist[grade];
        sum += grade * hist[grade];
    }
    return ((double)sum)/count;
}

/* Histogramm als Tabelle ausgeben.

```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```

*/
void showHist(int hist[]) {
    printf("#Zensur Anzahl\n");
    for (int grade = 0; grade <= MAX_GRADE; grade++) {
        printf("%d %2d\n", grade, hist[grade]);
    }
}

int main() {
    int hist[MAX_GRADE+1]; // Zensuren 1...5, hist[0] für ungültige Werte
    int count = readHist(hist);
    double mean = meanHist(hist);
    // Ausgabe als Gnuplot-Kommentar
    printf("# Anzahl %d, Durchschnitt: %f\n", count, mean);
    showHist(hist);
}
    
```

Werden mehrdimensionale Felder an Funktionen übergeben, müssen alle Dimensionen bis auf die erste Dimension spezifiziert sein. Sind das zur Compile-Zeit bekannte Konstanten (int-Literale oder enum-Werte), können wir diese einfach in eckigen Klammern angeben.

Beispiel: Lösen eines Gleichungssystems mit 2 Unbekannten. Wir definieren die benötigten Felder **a** für die Koeffizienten, **b** für die rechte Seite und **x** für die Lösung. Für das Einlesen der Daten, und Lösen der Gleichung benutzen wir Funktionen und nutzen aus, dass die Felder als Referenz übergeben werden. Die Funktionen für die Anzeige des Gleichungssystems und die Probe verändern dagegen die Felder nicht.

Quellcode 7.5: lineq2.c: LGS mit 2 Unbekannten

```

/* Gleichungssystem mit 2 Unbekannten.
 * Eingabe von Konsole:
 * a11 a12 b1
 * a21 a22 b2
 */

#include <stdio.h>

double readDouble() {
    double res;
    scanf("%lf", &res);
    return res;
}

void readData(double a[2][2], double b[2]) {
    for (int row = 0; row < 2; row++) {
        for (int col = 0; col < 2; col++) {
            a[row][col] = readDouble();
        }
        b[row] = readDouble();
    }
}

void showEq(double a[2][2], double b[2]) {
    for (int row = 0; row < 2; row++) {
        printf("%5.1f x1 + %5.1f x2 = %5.1f\n", a[row][0], a[row][1], b[row]);
    }
}
    
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```

void solve2(double a[2][2], double b[2], double x[2]) {
    // erste Zeile *a21, zweite Zeile *a11
    // dann 1. von 2. Zeile abziehen (1. Spalte wird 0)
    // Ergebnis: aa x2 = bb
    double aa = a[0][0] * a[1][1] - a[1][0] * a[0][1];
    double bb = a[0][0] * b[1] - a[1][0] * b[0];
    // Berechne x2 = b2/a2
    x[1] = bb/aa;
    // Berechne x1 = (b1 - a12 x2)/a11
    x[0] = (b[0] - a[0][1]*x[1])/a[0][0];
}

/* Quadrat einer Zahl. */
double sqr(double x) {
    return x*x;
}

/* Probe
 * :return: Quadratischer Fehler
 */
double eval2(double a[2][2], double x[2], double b[2]) {
    double err = 0.0;
    for (int row = 0; row < 2; row++) {
        double sum = 0.0;
        for (int col = 0; col < 2; col++) {
            sum += a[row][col] * x[col];
        }
        err += sqr(sum - b[row]);
    }
    return err;
}

int main() {
    double a[2][2];
    double b[2];
    double x[2];

    readData(a, b);
    showEq(a, b);
    // Lösen, Ergebnis in x[]
    solve2(a, b, x);
    // Probe
    double err = eval2(a, x, b);
    printf("Ergebnis: x1 = %5.1f, x2 = %5.1f, Fehler: %f\n", x[0], x[1], err);
}
    
```

Download: lineq2.c, Beispieldaten lineq2-2.dat, Ergebnis:

```

3.0 x1 + 2.0 x2 = 1.0
4.0 x1 + 5.0 x2 = 6.0
Ergebnis: x1 = -1.0, x2 = 2.0, Fehler: 0.000000
    
```

Wollen wir die Dimensionen dagegen erst zur Laufzeit festlegen, dann können wir für die Dimensionsangabe `int`-Parameter verwenden, die jedoch in der Parameterliste der Funktion *vorher* aufgeführt werden müssen:

```
int f(int rows, int cols, double a[rows][cols]) {
    // ...
}
```

Beispiel: Erzeugen eines einfachen lateinischen Quadrats. Die gewünschte Größe kann beim Aufruf angegeben werden.

Quellcode 7.6: latSquare.c: Lateinisches Quadrat variabler Größe

```
/* Triviales lateinisches Quadrat.
 * Funktionsruf mit variabler mehrdimensionaler Feldgröße.
 * Größe auf Kommandozeile.
 */

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

void newLatSquare(int size, int square[size][size]) {
    for (int row = 0; row < size; row++) {
        for (int col = 0; col < size; col++) {
            square[row][col] = 1 + (row + col) % size;
        }
    }
}

void showSquare(int size, int square[size][size]) {
    for (int row = 0; row < size; row++) {
        for (int col = 0; col < size; col++) {
            printf("%3d", square[row][col]);
        }
        printf("\n");
    }
}

int main(int argc, char* argv[]) {
    int size = 5;
    if (argc == 2) {
        size = atoi(argv[1]);
    }
    int square[size][size];
    newLatSquare(size, square);
    showSquare(size, square);
}
```

Download: latSquare.c.

7.6 Einfügen und Löschen von Feldelementen

Häufig werden wir Felder als *Container* benutzen, um während des Programmlaufs neue Werte aufnehmen und andere Werte wieder löschen zu können. In solchen Szenarien definieren wir unser Feld mit einer maximal zu erwartenden Größe und füllen es nur zum Teil mit unseren benötigten Werten.

Um die Zahl der aktuell vorhandenen Feldelemente feststellen zu können, bieten sich folgende zwei Möglichkeiten an:

Verwenden eines Stoppelements. Ein bestimmtes Datum (None-Wert) wird nicht als echter Wert benötigt und wird als letztes Feldelement angehängt. Wir ermitteln die Feldlänge, indem wir die

Positions dieses Stoppelements suchen.

Getrenntes Speichern der Feldlänge. Wir merken uns die tatsächliche Feldlänge in einer Hilfsvariablen, die beim Hinzufügen und Löschen von Elementen aktualisiert werden muss.

Bei Gleitpunktzahlen können wir die erste Variante verwenden, indem wir den Spezialwert NaN als Stoppelement verwenden. Bei `Strings` wird in C prinzipiell so verfahren. Für `int`-Werte können wir meist nicht so vorgehen, da hier jeder mögliche Wert eine sinnvolle Zahl darstellt. Hier werden wir meist die zweite Form verwenden.

Wir beispielsweise unsere benötigten Daten wie folgt definieren:

```
const int CAP = 1000; // maximale Feldkapazität

int main() {
    int list[CAP];
    int len = 0; // aktueller Füllstand
    // ...
}
```

Das Anfügen eines Feldelements kann in einer Funktion folgendermaßen realisiert werden (hier ohne Prüfung der Kapazität):

```
int appendVal(int val, int list[], int len) {
    list[len] = val;
    return len+1;
}
```

Die Länge ist gleichzeitig der Index des ersten freien Feldelements (das `len`-te Element steht an Position `len-1`). Wir müssen unbedingt die Länge aktualisieren, deshalb geben wir die neue Länge zurück und könnten die Funktion folgendermaßen nutzen:

```
len = appendVal(42, list, len);
```

Wollen wir ein Element nicht am Ende, sondern an einer Position zwischen bereits genutzten Elementen einfügen, müssen wir erst alle Elemente ab dieser Position um eine Position nach hinten schieben, indem wir jedes Feldelement auf seinen nächsthöheren Index kopieren.

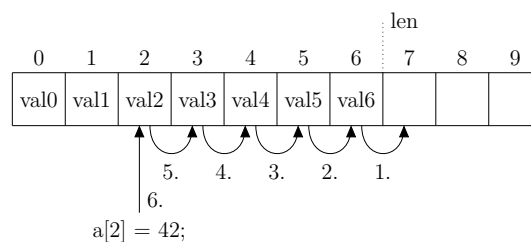


Abb. 7.5: Einfügen eines Elements

Wir müssen dabei die Reihenfolge des Umkopierens beachten. Beginnen wir an Position 2 und kopieren das Element auf Position 3, dann wird der Wert an Position 3 unwiderruflich überschrieben. So setzt sich das Überschreiben bis ans Ende fort, alle verschobenen Feldelement haben nun den alten Wert von Position 2. Wir beginnen deshalb rechts und kopieren zunächst das letzte Element auf die erste freie Position. Dieses ist nun gesichert, wir können das vorletzte Element auf die alte Position des letzten Elements kopieren etc.

Zum Löschen eines Elements kopieren wir einfach alle folgenden Elemente eine Position nach links.

Hier müssen wir mit dem Umkopieren an der Löschposition beginnen und uns schrittweise nach rechts bis zum Ende durcharbeiten.

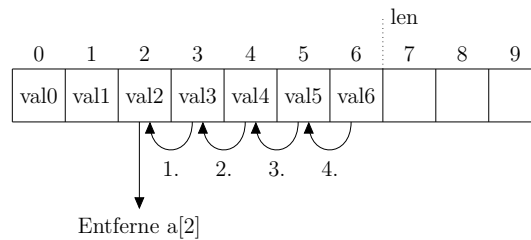


Abb. 7.6: Löschen eines Elements

Beispiel: Einlesen einer mit -1 terminierten Zahlenfolge in ein sortiertes Feld und anschließendes Löschen aller ungeraden Werte. Unser Feld ist zu jedem Zeitpunkt sortiert. Um ein neues Element aufzunehmen, suchen wir die erste Feldposition, die einen größeren als den neuen Wert enthält, und fügen das neue Element an dieser Stelle ein. Beim Löschen müssen wir ebenfalls aufpassen: Es verschieben sich die Positionen aller Werte nach dem gelöschten Element. Wir löschen deshalb jeweils nur das erste Element, das der Löschbedingung entspricht, und wiederholen diesen Schritt solange, bis kein solches Element mehr gefunden wird.

Quellcode 7.7: sortinsert.c: Sortiertes Einfügen und Löschen ungerader Werte

```

/* Sortiertes Einfügen von Werten in ein Feld.
 * Anschließend alle ungeraden Werte löschen.
 * Eingabe von Konsolen, Abschluss -1
 */

#include <stdio.h>
#include <assert.h>

const int CAP = 1000; // Feldkapazität

int readInt() {
    int res;
    scanf("%d", &res);
    return res;
}

/* Suche Position, an der val ins Feld eingefügt werden muss.
 * :return: Einfügeposition
 */
int insertPos(int val, int a[], int len) {
    int i = 0;
    while (i < len) {
        if (val < a[i]) return i;
        i++;
    }
    assert(i == len);
    // alle Werte kleiner -> am Ende anhängen
    return len;
}

/* Wert an Position einfügen.
 * :return: Neue Länge
 */
int insVal(int val, int pos, int a[], int len) {
    // Platz schaffen

```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```

    for (int i = len; i > pos; i--) {
        a[i] = a[i-1];
    }
    // Wert einfügen
    a[pos] = val;
    return len+1;
}

/* Suche erste ungerade Zahl.
 * :return: Position oder -1 (nicht gefunden)
 */
int firstOdd(int a[], int len) {
    for (int i = 0; i < len; i++) {
        if (a[i] % 2 != 0) return i;
    }
    // nicht gefunden
    return -1;
}

/* Löschen des Elements an Position.
 * :return: neue Feldlänge
 */
int delVal(int pos, int a[], int len) {
    if (pos >= len) return len;
    // Elemente eine Position nach vorn schieben
    for (int i = pos; i < len-1; i++) {
        a[i] = a[i+1];
    }
    return len-1;
}

/* Feldanzeige.
 */
void showList(int a[], int len) {
    printf("Liste:");
    for (int i = 0; i < len; i++) {
        printf(" %d", a[i]);
    }
    printf("\n");
}

int main() {
    int a[CAP];
    int len = 0;
    int val = readInt();
    while (val != -1 && len < CAP) {
        int pos = insertPos(val, a, len);
        len = insVal(val, pos, a, len);
        val = readInt();
    }
    showList(a, len);
    // alle ungeraden Elemente löschen
    int pos = firstOdd(a, len);
    while (pos >= 0) {
        len = delVal(pos, a, len);
        pos = firstOdd(a, len);
    }
}

```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
}
    showList(a, len);
}
```

7.7 Aufgaben

Aufgabe 7.1 ISBN

Schreiben Sie eine Funktion, die eine 13-stellige ISBN auf ihre Korrektheit überprüft: Dazu werden von vorn nach hinten alle Ziffern an ungeraden Positionen addiert, alle Ziffern an geraden Positionen werden mit 3 multipliziert und dann ebenfalls addiert. Diese Summe muss durch 10 teilbar sein.

Da eine solche Zahl (derzeit) zu groß für den Typ `int` ist, lesen Sie sie ziffernweise (durch Leerzeichen getrennt) von der Konsole in ein

Aufgabe 7.2 Reiskörner

Der Legende nach wurde der Erfinder des Schachspiels vom Herrscher nach der gewünschten Belohnung gefragt. Er wünschte sich etwas Reis, und zwar ein Reiskorn für das erste, zwei Reiskörner für das zweite und immer wieder die doppelte Zahl von Reiskörnern für das folgende Feld. Schreiben Sie ein Programm, dass die Belohnung des Erfinders berechnet. Rechnen Sie die Menge in Kilogramm um, wobei ein einzelnes Reiskorn 30 mg wiegt.

Hinweis: Prüfen Sie die Zwischenergebnisse!

Aufgabe 7.3 Game of Life

Programmieren Sie das Spiel *Life* von Conway: Die Lebewesen leben in quadratischen Feldern eines Blattes. Lebewesen mit 0 oder 1 Nachbarn sterben in der nächsten Generation an Isolation, Lebewesen mit 4 oder mehr Nachbarn an Überbevölkerung. In leeren Zellen mit genau 3 Nachbarn wird in der nächsten Generation ein Lebewesen geboren.

Wählen Sie das Spielfeld mit fester Größe und lesen sie die Startpopulation nach folgendem Muster ein:

```
generationenZahl
anzahl
x1 y1
x2 y2
...
```

Erzeugen Sie nach jeder neuen Generartion sinnvollerweise eine Ausgabe.

Aufgabe 7.4 Quadratische Matrizen

Erzeugen Sie eine quadratische Matrix einer durch eine Eingabe bestimmten Größe (maximal 100 Zeilen/Spalten) mit folgendem Aufbau (Bsp. $n=4$):

```
1  2  4  7
3  5  8 11
6  9 12 14
10 13 15 16
```

Die natürlichen Zahlen sollen also der Reihe nach diagonal nach links unten aufgefüllt werden.

Die Größe der Matrix wird also Kommandozeilenparameter vorgegeben.

- Schreiben Sie eine Funktion zur Anzeige der Matrix.

- Schreiben Sie eine Funktion zum Spiegeln der Matrix an der Gegendiagonale.
- Schreiben Sie eine Funktion zum Rotieren der Matrix im Uhrzeigersinn. Ergebnis für obiges Beispiel:

10	6	3	1
13	9	5	2
15	12	8	4
16	14	11	7

- Schreiben Sie eine Funktion, die die Summe der Zahlen in allen vier Dreiecken ohne die Diagonalen sowie die Summe der Diagonalen berechnet und anzeigt. Im Beispiel: Summe-Nord=6, Summe-Ost=25, Summe-Süd=28, Summe-West=9, Summe-Hauptdiagonale=34, Summe-Gegendiagonale=34

Schreiben Sie ein Verwaltungsprogramm, das den Aufruf aller Funktionen testet.

Aufgabe 7.5 *Matrixoperationen*

Schreiben Sie Funktionen zur Multiplikation zweier Vektoren (inneres Produkt), Multiplikation von Matrix und Vektor und Matrix und Matrix. Schreiben Sie dazu geeignete Testprogramme, die Testmatrizen bzw. -vektoren per Dateiumlenkung aus Dateien lesen.

Aufgabe 7.6 *Magisches Quadrat*

Schreiben Sie eine Funktion, die überprüft, ob ein gegebenes zweidimensionales Feld ein magisches Quadrat ist. Die maximale Feldgröße soll 100×100 betragen, die tatsächliche Feldgröße ist als Parameter zu übergeben. Lesen Sie das Quadrat von der Kommandozeile oder besser aus einer Datei per Dateiumlenkung) in folgender Form ein (# bezeichnen Kommentare, die nicht in der Eingabe enthalten sind):

4				# Groesse
16	5	9	4	# 1. Zeile
2	11	7	14	# 2. Zeile
3	10	6	15	# 3. Zeile
13	8	12	1	# 4. Zeile

Download dieses Beispiels: `magic1.dat`.

Aufgabe 7.7 *Magisches Quadrat erzeugen*

Schreiben Sie ein Programm, das magische Quadrate mit vom Nutzer wählbarer, *ungerader* Größe nach folgendem Algorithmus erzeugt ([Beschreibung und Demo](#) – Siamesische Methode):

- Schreibe die 1 in die erste Zeile in die Mitte.
- Gehe nun einen Schritt nach rechts und nach oben und schreibe dort die nächste Zahl hin.
- Wird dabei das Feld verlassen, betreten wir es am entgegengesetzten Rand an der gleichen Stelle (wir denken uns also das Quadrat an den gegenüberliegenden Kanten verklebt).
- Ist das nächste zu füllende Feld schon belegt, gehen wir nicht nach rechts oben, sondern geradeaus nach unten und schreiben dort die nächste Zahl hin. Sind alle Felder gefüllt, brechen wir ab.

Testen Sie auch Ihr selbst erzeugtes Quadrat, ob es korrekt ist.

- *Ziel*
- *Definition und Verwendung*
- *Strukturen und Funktionen*
- *Ein- und Ausgabe*
- *Structs und Felder*
- *Aufgaben*

8.1 Ziel

Mit Feldern können wir eine größere Zahl gleichartiger Variablen zu einer Einheit zusammenfassen. Häufig wollen wir aber auch Daten von unterschiedlichen Typen als Einheit betrachten. Wollen wir z. B. eine Lehrveranstaltung betrachten, dann besteht sie u. a. aus folgenden Komponenten:

- Bezeichnung (Text)
- Typ (V, U, P, ... – Zeichen)
- Dauer (Zahl)
- Wochentag (Mo, Di, Mi, ... – als Zahl codierbar)
- Zeit (Zeittyp, auch als Zahl codierbar)

Wollen wir eine Anzahl von Veranstaltungen in Feldern verwalten, müssten wir nach den bisherigen Kenntnissen für jede Komponente ein eigenes Feld definieren und bei Änderungen alle Felder parallel in der gleichen Form anpassen. Da dies sehr aufwendig und fehleranfällig ist, benötigen wir einen neuen Datentyp, der die notwendigen Komponenten zu einer Einheit zusammenfasst, die wir dann z. B. in einem Feld ablegen können. Man nennt dieses Konzept *Datenkapselung*, das uns später in erweiterter Form als *Klasse* in der *Objektorientierten Programmierung* wieder begegnen wird.

8.2 Definition und Verwendung

Wir müssen dazu zunächst deklarieren, welche Komponenten unser Struct enthalten soll. Das sollte außerhalb aller Funktionen am Anfang unseres Programms oder in einem eigenen Headerfile geschehen:

```
struct StructName {
    typ komponente;
    ...
};
```

Warnung: Wir dürfen das Semikolon nach der letzten geschweiften Klammer nicht vergessen!

Beispiel: Bruch, der Zähler, Nenner und seine Fließpunktrepräsentation enthält.

```
struct Fraction {
    int nom;
    int denom;
    double val;
};
```

Jede Komponente wird wie eine einfache Variable des entsprechenden Typs definiert, allerdings ist eine Initialisierung nicht zulässig. Ist einmal unser Struct-Typ deklariert, können wir damit innerhalb unserer Funktionen Variablen definieren:

```
struct StructName variable [= {initWert1, ...}];
```

Die Initialisierungswerte werden den Komponenten in der Reihenfolge der Definition zugeordnet. Alternativ können wir eine gezielte Zuweisung durch Angabe des Komponentennamens erreichen:

```
.komponente = wert
```

Beispiel. Um mit unserem Struct `Fraction` Variablen zu definieren, schreiben wir in einer Funktion (z. B. `main()`):

```
int main() {
    struct Fraction frac1; // uninitialisiert
    struct Fraction frac2 = { 1, 2, 0.5}; // initialisiert
    struct Fraction frac3 = {
        .nom = 4,
        .denom = 5;
    }; // Initialisierung mit Komponentennamen, .val ist nicht initialisiert
}
```

Die Komponenten einer Struktur verhalten sich wie Variablen des gleichen Typs. Wir greifen auf sie über das Konstrukt

```
structVariable.komponente
```

zu.

Beispiel: Setzen des Fließpunktwertes des Bruchs `frac3`:

```
frac3.val = (double)frac3.nom/frac3.denom;
```

8.3 Strukturen und Funktionen

Im Gegensatz zu Feldern werden Strukturen an Funktionen als *Wert* übergeben. Eine Veränderung innerhalb der Funktion wirkt sich also nicht auf den Struct in der rufenden Funktion auf. Ebenso können

Funktionen Structs als Rückgabewert verwenden.

Beispiel: Um den Fließpunktwert unseres Bruchs nicht manuell berechnen zu müssen, schreiben wir eine Funktion, die einen vollständig initialisierten Bruch allein aus Zähler und Nenner konstruiert. Dieser wird auch so normalisiert, dass der Nenner stets größer 0 ist. Eine zweite Funktion `showFraction()` zeigt einen Bruch an.

```

/* Struct Fraction mit Funktion zum Erzeugen
 */

#include <stdio.h>

struct Fraction {
    int nom;    // Zähler
    int denom;  // Nenner
    double val; // float-Wert
};

/* Erzeugen Bruch aus Zähler und Nenner, Setzen von val.
 * :return: Fraction-Objekt
 */

struct Fraction newFraction(int nom, int denom) {
    struct Fraction res = {
        .nom = nom,
        .denom = denom
    };
    // Normalisierung
    if (res.denom < 0) {
        res.nom = -res.nom;
        res.denom = -res.denom;
        // TODO: Kürzen
    }
    res.val = (double)res.nom/res.denom;
    return res;
}

void showFraction(struct Fraction frac) {
    printf("Fraction %d/%d (%f)\n", frac.nom, frac.denom, frac.val);
}

int main() {
    struct Fraction frac1 = {-2, -3, 1.5}; // nicht normalisiert, val falsch
    struct Fraction frac2 = newFraction(3, 5);
    struct Fraction frac3 = newFraction(4, -6);
    showFraction(frac1);
    showFraction(frac2);
    showFraction(frac3);
}

```

Ergebnis:

```

Fraction -2/-3 (1.500000)
Fraction 3/5 (0.600000)
Fraction -4/6 (-0.666667)

```

8.4 Ein- und Ausgabe

Eine direkte Ein- oder Ausgabe von Structs ist in C nicht möglich, wir müssen hier jede Komponente einzeln behandeln. Wie wir die Ausgabe realisieren könnten, haben wir oben bereits gesehen.

Für die Eingabe eines Structs schreiben wir am besten eine Funktion, die mit Hilfe unserer Eingabefunktionen `readInt()` und `readDouble()` die Werte aller Komponenten der Struktur einliest und als Ergebnis eine Struct-Variable zurückgibt.

Beispiel: Einlesen einer Anzahl von Brüchen, Abbruch durch die Eingabe von 0 0.

Quellcode 8.1: `fraction2.c`: Brüche mit Eingabe, Abbruch mit ungültigem Bruch, Addition aller Brüche

```
/* Einlesen von Brüchen bis zu ungültiger Eingabe (Nenner 0)
 * und Addition aller Brüche.
 */

#include <stdio.h>
#include <stdbool.h>

int readInt() {
    int res;
    scanf("%d", &res);
    return res;
}

struct Fraction {
    int nom;    // Zähler
    int denom;  // Nenner
    double val; // float-Wert
};

/* Erzeugen Bruch aus Zähler und Nenner, Setzen von val.
 * :return: Fraction-Objekt
 */
struct Fraction newFraction(int nom, int denom) {
    struct Fraction res = {
        .nom = nom,
        .denom = denom
    };
    // Normalisierung
    if (res.denom < 0) {
        res.nom = -res.nom;
        res.denom = -res.denom;
        // TODO: Kürzen
    }
    // Nulldivision liefert NaN
    res.val = (double)res.nom/res.denom;
    return res;
}

/* Prüfen, ob Bruch gültig ist (Nenner != 0)
 */
bool isValidFraction(struct Fraction frac) {
    return frac.denom != 0;
}
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```

/* Addition zweier Brüche.
 * :param f1, f2: Summanden
 * :return: Summe als Bruch
 */
struct Fraction addFractions(struct Fraction f1, struct Fraction f2) {
    int denom = f1.denom * f2.denom;
    int nom = f1.nom * f2.denom + f2.nom * f1.denom;
    return newFraction(nom, denom);
}

/* Einlesen eines Bruchs von der Konsole.
 * Reihenfolge: Zähler Nenner
 */
struct Fraction readFraction() {
    int nom = readInt();
    int denom = readInt();
    return newFraction(nom, denom);
}

void showFraction(struct Fraction frac) {
    printf("Fraction %d/%d (%f)\n", frac.nom, frac.denom, frac.val);
}

int main() {
    struct Fraction frac = readFraction();
    struct Fraction sum = newFraction(0, 1); // Bruch mit Wert 0
    while (isValidFraction(frac)) {
        showFraction(frac);
        sum = addFractions(sum, frac); // sum += frac
        frac = readFraction();
    }
    // Debug: Ausgabe Abbruchwert
    printf("Abbruchwert: ");
    showFraction(frac);
    printf("Summe: ");
    showFraction(sum);
}

```

Download: fraction2.c. Beispielergebnis:

```

1 2
2 3
3 4
0 0

```

Download: fraction2.dat. Ergebnis:

```

Fraction 1/2 (0.500000)
Fraction 2/3 (0.666667)
Fraction 3/4 (0.750000)
Abbruchwert: Fraction 0/0 (-nan)
Summe: Fraction 46/24 (1.916667)

```

Bemerkung: Wir haben mehrere Funktionen definiert, die in dieser Form nur für den Datentyp `struct Fraction` sinnvoll sind. Es wäre deshalb schön, wenn man diese Funktionen ebenso wie die Daten in der Datenstruktur zusammenfassen könnte. Diese Möglichkeit wurde mit dem Konzept der *OOP* geschaffen,

das wir uns in Python (*Python-Klassen*) anschauen werden.

8.5 Structs und Felder

Wie schon oben gesagt, wird das Kapseln zusammengehöriger Daten immer dann wichtig, wenn wir viele Daten dieses Typs gemeinsam verwalten wollen. Für solche Aufgaben kennen wir bisher nur Felder. Diese können nicht nur Elemente von einfachen Datentypen, sondern auch vom Struct-Typ enthalten. Wir vereinbaren diese nach dem gleichen Muster wie folgt:

```
struct StructurName fieldName[größe];
```

Eine Initialisierung ist ebenfalls möglich. Die Liste der Feldelemente wird in geschweifte Klammern eingeschlossen, die Werte jedes Structs ebenfalls.

Durch die Angabe von

```
fieldName[index]
```

greifen wir auf den kompletten Struct zu (um ihn z. B. einer Funktion zu übergeben), mit der Angabe

```
fieldName[index].komponente
```

erreichen wir die Komponente des Structs, der im Feld an der Position *index* steht.

Beispiel: Initialisiertes Feld von Brüchen. Die Größe wird automatisch aus der Initialisierung abgeleitet. Um die Länge nicht speichern zu müssen, ist der letzte Feldeintrag ein Stoppelement, wofür wir einen ungültigen Bruch verwenden.

Quellcode 8.2: fraction3.c: Feld von Brüchen, initialisiert, mit
Stoppelement

```
/* Feld von Brüchen, nun ohne Fließpunktinterpretation */
#include <stdio.h>
#include <stdbool.h>

struct Fraction {
    int nom;    // Zähler
    int denom;  // Nenner
};

bool isValidFraction(struct Fraction frac) {
    return frac.denom != 0;
}

void showFraction(struct Fraction frac) {
    printf("Fraction %d/%d\n", frac.nom, frac.denom);
}

int main() {
    // Ende des Feldes durch Bruch 0/0 gekennzeichnet
    struct Fraction fractions[] = {
        {3, 4},
        {2, 2},
        {5, 8},
        {0, 0},
    };
    // Manuelle Ausgabe 1. Bruch
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
printf("Zaehler: %d, Nenner %d\n", fractions[0].nom, fractions[0].denom);
// Ausgabe
for (int i = 0; isValidFraction(fractions[i]); i++) {
    showFraction(fractions[i]);
}
}
```

Umgedreht können Structs auch Felder enthalten, allerdings muss nun die Feldgröße zur Übersetzungszeit bekannt sein. Wir müssen sie also als Literal angeben oder eine Konstante mit dem Typ `enum` (s. *Konstanten*) definieren.

Wenn wir Felder als Container für Datensammlungen variabler Größe verwenden, ist es sinnvoll, das Feld gemeinsam mit der tatsächlich genutzten Größe zu verwalten. Mit unseren bisherigen Mitteln müssen wir allerdings eine Maximalgröße für alle Felder zur Compilezeit festlegen.

Beispiel: Einlesen von maximal 1000 Werten in ein Feld (Abschluss mit 0) und anschließendes Aufteilen in zwei Felder mit allen Werten kleiner und allen Werten größer des Mittelwertes.

Quellcode 8.3: splitmean.c: Zerlegen einer Liste am Mittelwert

```
/* Einlesen eines Feldes, max. 1000 Werte, Abschluss mit 0. Zerlegen
 * in zwei Felder mit allen Werten kleiner bzw. größer des Mittelwerts.
 */
#include <stdio.h>

enum { CAP = 1000 }; // Maximale Kapazität der Listen

struct List {
    int val[CAP];
    int len;
};

int readInt() {
    int res;
    scanf("%d", &res);
    return res;
}

/* Erzeugen einer leeren Liste.
 * :return: Liste
 */
struct List newList() {
    struct List res;
    res.len = 0;
    return res;
}

/* Anhängen eines Elements an eine Liste.
 * :param list: alte Liste
 * :param val: neuer Wert
 * :return: neue Liste (Kopie)
 */
struct List listAppend(struct List list, int val) {
    if (list.len >= CAP) return list; // Liste ist voll
    list.val[list.len] = val;
    list.len++;
    return list;
}
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```

}

/* Einlesen der Liste bis Abbruchwert 0.
 * :return: Liste
 */
struct List readList() {
    struct List res = newList();
    int val = readInt();
    while (val != 0) {
        res = listAppend(res, val);
        val = readInt();
    }
    return res;
}

/* Anzeige einer Liste.
 * :param list: Liste
 */
void showList(struct List list) {
    for (int i = 0; i < list.len; i++) {
        printf(" %d", list.val[i]);
    }
    printf("\n");
}

/* Mittelwert der Liste (ganzzahlig).
 * :param list: Liste
 * :return: Mittelwert
 */
int listMean(struct List list) {
    int sum = 0;
    // Nulldivision vermeiden
    if (list.len == 0) return 0;
    for (int i = 0; i < list.len; i++) {
        sum += list.val[i];
    }
    return sum/list.len;
}

/* Liste aller Werte <= Vorgabewert.
 * :param list: Liste
 * :param split: Splitwert
 * :return: neue Liste
 */
struct List listSmaller(struct List list, int split) {
    struct List res = newList();
    for (int i = 0; i < list.len; i++) {
        if (list.val[i] <= split) {
            res = listAppend(res, list.val[i]);
        }
    }
    return res;
}

/* Liste aller Werte > Vorgabewert.
 * :param list: Liste

```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```

* :param split: Splitwert
* :return: neue Liste
*/
struct List listLarger(struct List list, int split) {
    struct List res = newList();
    for (int i = 0; i < list.len; i++) {
        if (list.val[i] > split) {
            res = listAppend(res, list.val[i]);
        }
    }
    return res;
}

int main() {
    struct List list = readList();
    int mean = listMean(list);
    struct List smaller = listSmaller(list, mean);
    struct List larger = listLarger(list, mean);
    printf("Eingabe: ");
    showList(list);
    printf("Mittelwert: %d\n", mean);
    printf("Kleinere Werte: ");
    showList(smaller);
    printf("Größere Werte: ");
    showList(larger);
}

```

 Download: `splitmean.c`.

8.6 Aufgaben

Aufgabe 8.1 *Zeiten*

Erweitern Sie die Struktur für Stunde und Minute um Tag und Woche und implementieren Sie Addition und Subtraktion von anderen Zeitstrukturen bzw. Minuten.

Aufgabe 8.2 *Brüche*

Erweitern Sie das Programm zur Bruchrechnung um das Kürzen, Subtrahieren Multiplizieren und Dividieren von Brüchen.

Erlauben Sie die Eingabe von Bruchrechenaufgaben mit zwei Brüchen, indem Sie die Rechenoperationen durch Zahlen codieren (+ = 1, - = 2, ...). Beispiel:

1	2	2	3	4
---	---	---	---	---

bedeutet $1/2 - 3/4$.

Aufgabe 8.3 *Quadratpuzzle, Hausaufgabe (8 Punkte)*

Ein Puzzle besteht aus quadratischen Teilen, die durch die Diagonalen in vier farbige Dreiecke zerlegt sind. Diese Quadrate dürfen nur so aneinandergelegt werden, dass sich gleichfarbige Dreiecke berühren. Die Farbe weiß (codiert mit dem Wert 0) passt dabei zu jeder Farbe.

Gegeben ist nun ein teilweise gelegtes Puzzle von rechteckiger Form, das noch einige Löcher enthält (gekennzeichnet durch ein Pseudoquadrat, dessen Dreiecke alle weiß sind – ein solches Teil gibt es bei

uns nicht im Spiel), sowie einige weitere, noch nicht verwendete Teile, mit denen die Löcher aufgefüllt werden sollen, falls das möglich ist. Diese noch nicht verbauten Teile können dabei in alle 4 Richtungen gedreht, aber nicht gespiegelt werden.

Die Eingabe soll wie folgt aussehen (# und der dahinterstehende Text stehen hier nur zur Erklärung und fehlen in der tatsächlichen Eingabe):

```
b h      # Breite Höhe
n o s w # Farben der Teile, zeilenweise
...
x        # Anzahl der zusätzlichen Teile
n o s w # Farben wie oben
...
```

Die erste Zeile enthält die Breite und Höhe des bereits ausgelegten Puzzles. Die folgenden $b * h$ Zeilen enthalten zeilenweise jeweils die Farben eines Quadrats in der Reihenfolge Nord (oben), Ost (rechts), Süd (unten), West (links). Vier Nullen bezeichnen eine Lücke. Es folgen eine Zeile mit der Anzahl der noch nicht verbauten Teile sowie zeilenweise die Beschreibung dieser Teile.

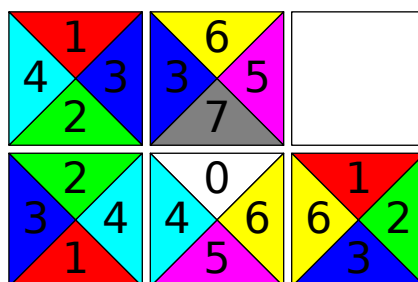
Die Farben sind durch Zahlen von 0 bis 9 codiert. Die maximale Größe des Puzzles ist 10×10 , die maximale Zahl der nichtverbauten Teile ist 20.

Beispieleingabe (Größe 2×3 , also 6 Teile im Puzzle, und 2 Ersatzteile):

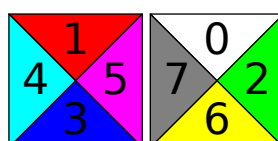
```
3 2
1 3 2 4
6 5 7 3
0 0 0 0
2 4 1 3
0 6 5 4
1 2 3 6
2
1 5 3 4
0 2 6 7
```

Entsprechendes Puzzle:

Puzzle:



Extrakacheln:



Beispiel-Puzzle und Ersatzkacheln

- Beschreiben Sie die Teile des Puzzles mit einem eigenen Struct-Datentyp, der die vier Farben als Einzelkomponenten beschreibt.
- Stellen Sie ausgelegte Puzzle als Feld dieser Struct-Objekte dar.
- Verwenden Sie für die nichtverbauten Teile ebenfalls ein Feld solcher Struct-Objekte.
- Lesen Sie das Puzzle und die nichtverbauten Teile aus einer Datei (mittels Dateiumlenkung) ein.
- Stellen Sie das Puzzle und die nichtverbauten Teile geeignet als *ASCII*-Graphik dar. Beispiel:

```
Puzzle:
+---+---+---+
|\1/|\6/|   |
|4 3|3 5|   |
|/2\|/7\|   |
|---+---+---+
|\2/|\0/|\1/|
|3 4|4 6|6 2|
|/1\|/5\|/3\|
|---+---+---+
```

```
Extrakacheln:
+---+ +---+
|\1/| |\0/|
|4 5| |7 2|
|/3\| |/6\|
+---+ +---+
```

- Schreiben Sie eine Funktion, die prüft, ob das bereits ausgelegte Puzzle korrekt ist (also nur gleiche Farben aneinanderstoßen). Die *main()*-Funktion soll diese Funktion aufrufen und eine Ausgabe der Form

```
Puzzle korrekt: ja
```

(bzw. *nein*) erzeugen.

- Schreiben Sie eine Funktion, die versucht, das Puzzle mit den nichtvorhandenen Teilen korrekt aufzufüllen. Geben Sie zurück, wieviele Teile Sie ergänzen konnten, und zeigen Sie das aufgefüllte Puzzle an. Die *main()*-Funktion soll diese Funktion aufrufen und ausgeben, wieviele Lücken gefüllt wurden und wie viele offen bleiben. Beispiel:

```
Luecken gefuehlt: 1, noch offen: 0
```

Sie können davon ausgehen, dass in den Testdaten jede Zusatzkachel nur in genau eine Lücke passt.

- Die Zusatzteile können gedreht, aber nicht gespiegelt werden, um die Lücken zu füllen.

Zeichen und Strings

- *Zeichentyp und -litterale*
- *Zeichenfelder und Strings*
- *Einlesen von Strings*
- *Stringfelder*
- *Einlesen von Stringfeldern*
- *Strings in Structs*
- *Aufgaben*

9.1 Zeichentyp und -litterale

Zur Darstellung von einzelnen Zeichen stellt *C* den Typ `char` zur Verfügung. Dabei handelt es sich ähnlich wie bei `bool` jedoch nicht um einen echten eigenen Typ, sondern lediglich einen ganzzahligen Typ mit der Speichergröße von 1 Byte = 8 Bit. Damit können also 256 verschiedene Zeichen dargestellt werden. Um die Arbeit mit Zeichen etwas bequemer zu machen, gibt es jedoch spezielle Zeichenlitterale:

`'ASCII-Zeichen'`

Damit wird der *ASCII*-Code des jeweiligen Zeichens bereitgestellt, z. B. 48 für das Zeichen 0 oder 65 für das Zeichen A. Dadurch können wir mit Zeichen einfach rechnen:

```
char c = 'A';           // 65
c = c + 2;              // 65, entspricht 'C'
int diff = '9' - '0';   // 9
```

Wir können also ganz einfach Klein- in Großbuchstaben umrechnen: wir bestimmen, wie weit der Buchstabe vom `a` entfernt ist und addieren die Differenz auf A.

```
char toUpper(char c) {
    // nur Kleinbuchstaben auswählen
    if (c >= 'a' && c <= 'z') {
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```

    return (c - 'a') + 'A';
} else {
    return c;
}
}

```

Warnung: Da die Länge einer `char`-Variable auf 1 Byte begrenzt ist, können viele Zeichen osteuropäischer Sprachen und nichtlateinische Buchstaben überhaupt nicht dargestellt werden. Die Darstellung von (deutschen) Umlauten ist nur dann möglich, wenn das Betriebssystem eine Latin-1-Codierung (z. B. ISO-8859-1) verwendet. Unter Linux wird normalerweise eine UTF-8-Codierung verwendet, bei der die Umlaute nicht mit einem Byte dargestellt werden.

Um `char`-Ausdrücke mit `printf()` auszugeben, benutzen wir das Muster `%c`. Beispiel:

```

char c = 'A';
printf("Zeichen: %c, Hexadezimalcode: %x Dezimalcode: %d\n",
      c, c, c);

```

Um nur ein einzelnes Zeichen auszugeben, können wir auch die Funktion `putchar()` verwenden:

```

char c = '\''; // Apostroph muss maskiert werden
putchar(c);

```

9.2 Zeichenfelder und Strings

Um mit ganzen Texten arbeiten zu können, verwenden wir Zeichenketten oder *Strings*. *C* besitzt allerdings keinen eigenen Stringtyp, sondern verwendet hierfür Zeichenfelder:

```
char string[maxSize];
```

Da wir bei der Bearbeitung des Strings häufig dessen Länge ändern wollen, benötigen wir eine Möglichkeit, die tatsächliche Stringlänge zu bestimmen. Das Mitführen einer Hilfsvariable ist recht umständlich, deshalb hängen wir an das Ende des Strings das Zeichen mit dem *ASCII*-Code 0 an (das `char`-Literal `c:type:'\0'`). Unser String benötigt dadurch eine Speicherzelle zusätzlich, was wir bei der Definition von Stringvariablen beachten müssen. Diese Konvention ist in *C* fest eingebaut. Wenn wir eine Stringvariable initialisieren wollen, geschieht das mit einem Stringliteral:

```
"zeichenfolge"
```

Wir können wie bei Zahlenfeldern die Größenangabe weglassen, wenn bei der Definition des Feldes eine Initialisierung erfolgt. Beispiele:

```

// char-Feld der Größe 10, Belegung: ['H', 'a', 'l', 'l', 'o', '\0', ...]
char s1[10] = "Hallo";
// char-Feld der Größe 5, Belegung: ['W', 'e', 'l', 't', '\0']
char s2[] = "Welt";

```

Da wir es hier wirklich nur mit Feldern von Zeichen zu tun haben, können wir auch das Feld auch mit den einzelnen Zeichen initialisieren, dürfen aber das Endekennzeichen nicht vergessen:

```

// identisch mit s2
char s3[] = {'W', 'e', 'l', 't', '\0'};

```

Hin und wieder finden wir in Beispielprogrammen folgende Syntax:

```
char *variable = "String";
```

Hier der String vom Compiler fest ins Programm eingebaut und darf oder kann nicht verändert werden. Wir sollten dies besser mit dem Modifikator `const` kennzeichnen:

```
const char * variable = "String";
```

Solange wir diesen String nur analysieren oder ausgeben und nicht verändern wollen, verhält er sich genauso wie die veränderlichen Strings.

Eine nachträgliche Zuweisung von Stringliteralen an Stringvariablen ist ebensowenig wie bei Zahlenfeldern zulässig. Folgender Code wird *nicht* funktionieren:

```
char s[10];
s = "Hallo"; // Funktioniert nicht!
```

Wir müssen statt dessen eine Funktion schreiben, die den Inhalt des einen in ein neues Feld zeichenweise kopiert:

```
enum {STR_MAX_LEN = 10};

void strCopy(char dst[], char src[], int maxLen) {
    int i; // Wert wird nach der Schleife noch benötigt
    for (i = 0; i < maxLen && src[i] != '\0'; i++) {
        dst[i] = src[i];
    }
    // Terminierungszeichen anhängen
    dst[i] = '\0';
}

int main() {
    char s[STR_MAX_LEN+1];
    strCopy(s, "Hallo", STR_MAX_LEN);
}
```

Diese Funktion stellt auch die C-Standardbibliothek unter dem Namen `strncpy()` (siehe: **man 3 strncpy**) bereit.

Für die Ausgabe von Stringvariablen mit `printf()` verwenden wir das Muster `%s`. Wir können hier eine Länge einfügen, der String wird linksbündig mit Leerzeichen aufgefüllt. Benutzen wir eine negative Längenangabe, erfolgt das Auffüllen rechtsbündig, wir können also einfach Tabellen bauen, bei denen Textspalten linksbündig ausgerichtet sind und an der gleichen Position enden, um die nächste Spalte zu beginnen.

Beispiel: Tabelle mit Farbwerten.

Quellcode 9.1: strout1.c Tabelle von Farbwerten

```
/* Ausgabe von Strings in Tabellenform
 */

#include <stdio.h>

int main() {
    char nameRed[] = "rot";
    int valRed = 0xff0000; // RGB-Wert rot
    char nameGreen[] = "gruen"; // Umlaut nicht zulässig
    int valGreen = 0x00ff00; // RGB-Wert grün
    char nameBlue[] = "blau";
    int valBlue = 0x0000ff; // RGB-Wert blau

    // Tabellenausgabe
    printf("%-10s %6s %10s\n", "Farbe", "Hex.", "Dezimal");
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
printf("%-10s %06x %10d\n", nameRed, valRed, valRed);
printf("%-10s %06x %10d\n", nameGreen, valGreen, valGreen);
printf("%-10s %06x %10d\n", nameBlue, valBlue, valBlue);
}
```

Ergebnis:

Farbe	Hex.	Dezimal
rot	ff0000	16711680
gruen	00ff00	65280
blau	0000ff	255

9.3 Einlesen von Strings

Unsere bisherigen Eingabefunktionen `readInt()` und `readDouble()` basieren auf der Funktion `scanf`. Sie liest die Zeichen der Eingabe immer bis zum Auftreten des nächsten *white space* (Leerzeichen, Tabulator oder Zeilenschaltung) und interpretiert den Wert. Da unsere Strings aber üblicherweise Leerzeichen enthalten, würden wir so immer nur das erste Wort einlesen. Wir lesen statt dessen besser die gesamte Zeile ein.

```
fgets(stringVariable, maxGröße, stdin)
```

Wir müssen dieser Funktion eine veränderliche Stringvariable (also den Namen eines `char`-Feldes) sowie die maximal erlaubte Länge übergeben. Unser Text wird nun bis zur maximalen Länge oder bis zum Zeilenende gelesen und im String gespeichert. Anschließend wird das `\0`-Zeichen angehängt. Unser Feld muss also ein Zeichen mehr als die maximale Länge aufnehmen können.

Der dritte Parameter beschreibt unseren Eingabestrom. Wir wollen uns hier nicht mit Dateiarbeit beschäftigen und verwenden immer die Konsole `stdin`.

Da das Zeilenendezeichen mit im String gespeichert wird, aber bei der weiteren Arbeit meist stört, schreiben wir eine Funktion, um es zu entfernen. Es kann nur einmal am Ende auftreten, deshalb genügt es, wenn wir es einfach mit `\0` überschreiben.

Beispiel: Einlesen des Strings und Anzeige, anschließend Entfernen des Newline-Zeichens. Um die Zeilenschaltung im String zu erkennen, wird er in der Ausgabe in `[]` eingeschlossen.

Quellcode 9.2: `readstr.c`: Einlesen eines Strings als Konsolenzeile.

```
/* Einlesen von Bewertungen in ein Feld.
 * Eingabe zeilenweise über Konsole bzw per Umlenkung aus Datei.
 */

#include <stdio.h>
#include <assert.h>

enum {
    MAX_STRLEN = 140, // max. Länge des Strings
};

/* Entferne gelesenes Newline-Zeichen.
 * Es kann max. eins als letztes Zeichen geben:
 * -> einfach durch \0 ersetzen
 */
void removeNl(char s[]) {
    int i = 0;
    while (s[i] != '\0' && s[i] != '\n') {
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```

    i++;
}
assert(s[i] == '\0' || s[i] == '\n');
s[i] = '\0';
}

/* Zeile einlesen.
*/
void readLine(char s[], int maxLen) {
    fgets(s, maxLen, stdin);
}

int main() {
    char line[MAX_STRLEN+1]; // Eingegebener String

    readLine(line, MAX_STRLEN);
    // Anzeige mit Newline
    printf("Eingelesene Zeile:\n");
    printf("[%s]\n", line);
    printf("Entferne Newline:\n");
    removeNl(line);
    printf("[%s]\n", line);
}

```

Ausgabe für die Konsoleneingabe **Hallo**Enter:

```

Eingelesene Zeile:
[Hallo
]
Entferne Newline:
[Hallo]

```

Später bauen wir den Aufruf von `removeNl()` direkt in die Funktion `readLine()` ein.

Warnung: In älteren Beispielen finden wir häufig die Funktion `gets()`. Diese sollte niemals benutzt werden, da beim Einlesen von Strings, die länger als die vorgesehene Feldgröße sind, sogenannte *Buffer Overflows* auftreten, die zu Abstürzen und Sicherheitslücken führen: Die überzähligen Zeichen überschreiben andere Werte in unserem Programmspeicher, z. B. auch die Rücksprungadresse einer Funktion, wodurch plötzlich völlig anderer Code ausgeführt wird.

Bemerkung: *C* stellt auch eine Funktion `getline()` bereit, die selbst den Speicher für den String reserviert. Dieser wird allerdings nicht automatisch freigegeben, wenn er nicht mehr benötigt wird. Wir werden deshalb diese Funktion hier nicht verwenden.

9.4 Stringfelder

Um eine Anzahl von Strings in einem Feld zu verwalten, benötigen wir wieder zweidimensionale Felder, da jeder String selbst schon ein Feld ist. Wir vereinbaren ein solches Feld wie folgt:

```
char fieldName[maxZahlStrings][maxStringLaenge] [Initialisierung];
```

Beispiel: Verwaltung der Ziffernamen (zehn Strings mit maximaler Länge 7 (6 Zeichen + Terminierung)).

Quellcode 9.3: strarray1: Stringfeld mit variablen Strings

```

/*
 * Feld von Strings
 * Formatierte Tabellenausgabe.
 */

#include <stdio.h>

int main() {
    char digitNames[10][7] = {
        "null",
        "eins",
        "zwei",
        "drei",
        "vier",
        "fuenf", // kein Umlaut möglich!
        "sechs",
        "sieben",
        "acht",
        "neun"
    };
    printf("%-10s %s\n", "Name", "Wert");
    for (int i = 0; i < 10; i++) {
        printf("%-10s %d\n", digitNames[i], i);
    }
    printf("Groesse der Datenstruktur: %ld byte\n", sizeof(digitNames));
}

```

Es entsteht ein Feld aus Zeichen mit genau 70 Zellen, wie wir uns mit der Anweisung `sizeof(variable)` überzeugen können. Um auf das zweite Zeichen des dritten Strings zuzugreifen, verwenden wir

```
digitNames[2][1]
```

Sollen die Strings unveränderlich sein und sind bereits zur Übersetzungszeit bekannt, können wir folgende alternative Definition verwenden:

```
const char *feldName[maxZahlStrings] = { :vr:`stringListe };
```

Beispiel: Ziffernnamen, die unveränderlich sind.

Quellcode 9.4: strarray2: Stringfeld mit konstanten Strings

```

/*
 * Feld von Strings
 * Formatierte Tabellenausgabe.
 */

#include <stdio.h>

int main() {
    const char *digitNames[10] = {
        "null",
        "eins",
        "zwei",
        "drei",
        "vier",
        "fuenf", // kein Umlaut möglich!
    };
}

```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```

        "sechs",
        "sieben",
        "acht",
        "neun"
    };
    printf("%-10s %s\n", "Name", "Wert");
    for (int i = 0; i < 10; i++) {
        printf("%-10s %d\n", digitNames[i], i);
    }
    printf("Groesse der Datenstruktur: %ld byte\n", sizeof(digitNames));
}

```

Bemerkung: Die interne Darstellung dieser zweiten Form unterscheidet sich erheblich von der ersten, wie wir an der Ausgabe von `sizeof()` sehen. Es wird ein eindimensionales Feld der angegebenen Größe erstellt, wobei jedes Feldelement einen Verweis (Pointer, siehe chap-pointer) auf den tatsächlichen Speicherort des Strings enthält. Die Verwendung ist jedoch in den meisten Fällen völlig identisch.

Diese Form benutzt auch die `main()`-Funktion zur Übergabe der Kommandozeilenparameter. Der erste Parameter `argc` enthält die Größe des Feldes `argv`, das die Verweise auf die einzelnen Strings enthält. Diese sind als unveränderlich anzunehmen (implementierungsabhängig), wie müssen den Kopf korrekt also so schreiben:

```
int main(int argc, const char * argv[])
```

9.5 Einlesen von Stringfeldern

Wollen wir eine Menge von Strings aus einer Datei einlesen, müssen wir die erste Form der Definition verwenden, da die Strings erst zur Laufzeit im Speicher abgelegt werden.

Beispiel: Einlesen von Bewertungen (Textzeilen) in ein Feld. Die Eingabe wird mit der Zeile `###` abgeschlossen.

Quellcode 9.5: `strarray3.c`: Zeilenweises Einlesen von Strings in Feld.

```

/* Einlesen von Bewertungen in ein Feld.
 * Eingabe zeilenweise über Konsole bzw per Umlenkung aus Datei.
 */

#include <stdio.h>
#include <stdbool.h>
#include <assert.h>

enum {
    MAX_STRLEN = 140, // max. Länge einer Zeile
    MAX_TEXTS = 100   // max. Zeilenzahl
};

/* Entferne gelesenes Newline-Zeichen.
 * Es kann max. eins als letztes Zeichen geben:
 * -> einfach durch \0 ersetzen
 */
void removeNl(char s[]) {

```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```

    int i = 0;
    while (s[i] != '\0' && s[i] != '\n') {
        i++;
    }
    assert(s[i] == '\0' || s[i] == '\n');
    s[i] = '\0';
}

/* Zeile einlesen und Newline entfernen.
 */
void readLine(char s[], int maxLen) {
    fgets(s, maxLen, stdin);
    removeNl(s);
}

/* String umkopieren in neues char-Feld.
 * :param dest: Zielstring
 * :param src: Ursprungsstring
 * :param maxLen: max. erlaubte Länge (ohne Stopzeichen)
 *
 * :see: man 3 strncpy
 */
int strCopy(char dest[], char src[], int maxLen) {
    int len;
    for (len = 0; len < maxLen && src[len] != '\0'; len++) {
        dest[len] = src[len];
    }
    assert(len < maxLen);
    dest[len] = '\0';
    return len;
}

/* Test auf Gleichheit zweier Strings.
 * :param s, t: Strings
 * :return: true, wenn Gleichheit
 */
bool strEq(char s[], char t[]) {
    int i;
    for (i = 0; s[i] != '\0' && t[i] != '\0'; i++) {
        if (s[i] != t[i]) return false;
    }
    // Entweder s[i] == t[i] == '\0' -> gleich
    // oder verschiedene Länge -> ungleich
    assert(s[i] == '\0' || t[i] == '\0');
    return s[i] == t[i];
}

void showTexts(char texts[MAX_TEXTS][MAX_STRLEN+1], int count) {
    for (int i = 0; i < count; i++) {
        printf("%3d [%s]\n", i, texts[i]);
    }
}

int main() {
    char texts[MAX_TEXTS][MAX_STRLEN+1];
    char line[MAX_STRLEN+1]; // für Eingabe

```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
int count = 0;
readLine(line, MAX_STRLEN);
while (!strEq(line, "###")) {
    strCopy(texts[count], line, MAX_STRLEN);
    count++;
    readLine(line, MAX_STRLEN);
}
showTexts(texts, count);
}
```

Beispieleingabe: texts.dat.

9.6 Strings in Structs

Wollen wir Strings in Structs verwenden, gehen wir wie bei Feldern innerhalb von Structs vor. Wir müssen eine feste maximale Länge definieren (mit `enum`) und fügen dann ein `char`-Feld als Komponente in den Struct ein. Solche Struct-Objekte können nun problemlos als Parameter und Rückgabewert einer Funktion verwendet werden (jeweils als Kopie).

Wollen wir solche Structs von der Konsole oder aus einer Datei (per Dateiumlenkung) einlesen, ist es wünschenswert, wenn alle Werte eines Objekts in einer Zeile stehen. Wir stehen also vor der Aufgabe, die Zeile in ihre Bestandteile zu zerlegen, wobei einzelne Bestandteile noch in Zahlen oder andere Datentypen umgewandelt werden müssen. Ein primitives Zerlegen selbst ist mit der Funktion `strtok()` (String Tokenizer) möglich.

Die Verwendung ist recht gewöhnungsbedürftig. Beim ersten Aufruf übergeben wir `strtok()` zwei Strings. Der erste String soll zerlegt werden, der zweite enthält hintereinander alle Zeichen, die als Trennzeichen benutzt werden soll.

Der Rückgabewert ist ein Verweis auf das erste *Token* (Stringabschnitt) bis zum ersten Trennzeichen. Weisen wir ihn einer Variablen zu, muss diese so vereinbart werden:

```
char * token;
```

Bei Folgeaufrufen ersetzen wir den ersten Parameter durch `NULL`. Das ist ein Makro mit dem Wert 0. Wir erhalten so das nächste Token. Dies wiederholen wir solange, bis der Rückgabewert der Funktion `NULL` ist.

Warnung: Bei Anwendung von `strtok()` wird der Ursprungsstring zerstört!

Beispiel: Zerlegen eines Beispielstrings in Wörter.

Quellcode 9.6: toktest.c: Zerlegung eines Strings

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

int main() {
    char text[] = "Die Antwort lautet: 42 ...";
    const char DELIMS[] = " .,:;!"; // Trennzeichen, hier Leer- und Satzzeichen
    char *token; // token[..] funktioniert nicht

    printf("Ursprungsstring: [%s]\n", text);
    token = strtok(text, DELIMS);
    while (token != NULL) {
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
printf("Token [%s]\n", token);
token = strtok(NULL, DELIMS);
}
printf("Ursungsstring zerstört: [%s]\n", text);
}
```

Ergebnis:

```
Ursungsstring: [Die Antwort lautet: 42 ...]
Token [Die]
Token [Antwort]
Token [lautet]
Token [42]
Ursungsstring zerstört: [Die]
```

Wir sehen hier bereits, dass leere Tokens übersprungen werden, sonst würden wir beim Verarbeiten der Punkte der Ellipse zwei leere Tokens erhalten.

Beispiel: Bewertung einer Lehrveranstaltung durch eine Zahl (1...5) und einen Text. Jede Bewertung ist ein Struct, der beide Daten zusammenfasst, alle Bewertungen werden in einem Feld verwaltet. Die Eingabe erfolgt durch zeilenweise, wobei Zahl und Text durch ^ getrennt werden. Die Eingabe wird durch die Zeile 0^### abgeschlossen.

Quellcode 9.7: scores.c: Bewertungen als Struct-Feld

```
/* Einlesen von einer Zahl von Bewertungen.
 * Jede Bewertung ist eine Zeile der Form: score^text
 * Abschluss mit 0^###
 */

#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <string.h>
#include <assert.h>

enum {
    MAX_STRLEN = 140,    // max. Länge einer Zeile
    MAX_SCORES = 100,    // max. Zeilenzahl
    MAX_SCORE_LEVEL = 5  // höchste Bewertung
};

/* Struktur für Bewertung */
struct Score {
    int level;
    char text[MAX_STRLEN + 1];
};

/* Entferne gelesenes Newline-Zeichen.
 * Es kann max. eins als letztes Zeichen geben:
 * -> einfach durch \0 ersetzen
 */
void removeNl(char s[]) {
    int i = 0;
    while (s[i] != '\0' && s[i] != '\n') {
        i++;
    }
}
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```

    assert(s[i] == '\0' || s[i] == '\n');
    s[i] = '\0';
}

/* Zeile einlesen und Newline entfernen.
 */
void readLine(char s[], int maxLen) {
    fgets(s, maxLen, stdin);
    removeNl(s);
}

/* String umkopieren in neues char-Feld.
 * :param dest: Zielstring
 * :param src: Ursprungsstring
 * :param maxLen: max. erlaubte Länge (ohne Stopzeichen)
 *
 * :see: man 3 strncpy
 */
int strCopy(char dest[], char src[], int maxLen) {
    int len;
    for (len = 0; len < maxLen && src[len] != '\0'; len++) {
        dest[len] = src[len];
    }
    assert(len < maxLen);
    dest[len] = '\0';
    return len;
}

/* Zeile in Struct einlesen
 */
struct Score readScore() {
    // Länge könnte größer gewählt werden, da Zahl abgeschnitten wird
    char line[MAX_STRLEN + 1];
    readLine(line, MAX_STRLEN);
    char *part0 = strtok(line, "^");
    char *part1 = strtok(NULL, "^");
    struct Score res;
    res.level = atoi(part0);
    strCopy(res.text, part1, MAX_STRLEN);
    return res;
}

/* Alle Zeilen einlesen in Feld.
 * :return: Anzahl Zeilen
 */
int readAllScores(struct Score scores[], int maxScores) {
    int len = 0;
    struct Score score = readScore(); // Testweise lesen
    while (len < maxScores && score.level != 0) {
        scores[len] = score;
        len++;
        score = readScore();
    }
    return len;
}

```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```

/* Ausgabe einer Bewertung.
 * Level wird durch *-Folge angezeigt.
 */
void showScore(struct Score score) {
    for (int i = 0; i < score.level; i++) {
        printf("*");
    }
    for (int i = score.level; i < MAX_SCORE_LEVEL; i++) {
        printf(" ");
    }
    printf(": %s\n", score.text);
}

/* Alle Bewertungen mit bestimmten Score-Wert ausgeben.
 * Bei level < 0 alle Bewertungen ausgeben.
 */
void showScores(struct Score scores[], int len, int level) {
    for (int i = 0; i < len; i++) {
        if (level == -1 || scores[i].level == level) {
            showScore(scores[i]);
        }
    }
}

/* Mittlere Bewertung.
 */
double meanScore(struct Score scores[], int len) {
    if (len == 0) return 0;
    int sum = 0;
    for (int i = 0; i < len; i++) {
        sum += scores[i].level;
    }
    return ((double)sum)/len;
}

int main() {
    struct Score scores[MAX_SCORES];
    int len = readAllScores(scores, MAX_SCORES);
    printf("Alle Bewertungen\n");
    showScores(scores, len, -1);
    printf("Beste Bewertungen\n");
    showScores(scores, len, 1);
    printf("Schlechteste Bewertungen\n");
    showScores(scores, len, MAX_SCORE_LEVEL);
    printf("Mittlere Bewertung: %f bei %d Bewertungen.\n",
        meanScore(scores, len), len);
}
    
```

Download: scores.c, Beispieldaten scores.dat. Ergebnis:

```

Alle Bewertungen
*      : Das war ja wohl nichts.
*****: Super.
*****: Grosse Klasse.
****  : Ganz ok.
***   : Ganz brauchbar.
    
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
*      : Grottenschecht.
**     : Gerade noch akzeptabel.
*****: Hervorragend.
Beste Bewertungen
*      : Das war ja wohl nichts.
*      : Grottenschecht.
Schlechteste Bewertungen
*****: Super.
*****: Grosse Klasse.
*****: Hervorragend.
Mittlere Bewertung: 3.250000 bei 8 Bewertungen.
```

Bemerkung: In Erweiterungsbibliotheken finden Funktionen, die auch komplexere Eingaben wie *CSV*- oder *JSON*-Dateien in ihre Komponenten zerlegen können. Wir werden diese im Folgesemester unter *Java* und *Python* verwenden.

9.7 Aufgaben

Aufgabe 9.1 *String-Standardfunktionen*

Informieren Sie sich über die Wirkungsweise der Funktionen `strlen()`, `strncat()` und `strcmp()`. Implementieren Sie diese Funktionen selbst (verwenden Sie andere Namen, um Compilerwarnungen zu vermeiden) und vergleichen Sie die Ergebnisse mit denen der Standardbibliothek.

Aufgabe 9.2 *Umwandlung in Großbuchstaben*

Schreiben Sie eine Funktion, die alle Zeichen eines Strings in Großbuchstaben umwandelt. Implementieren Sie zwei Versionen:

- In Place: der Originalstring wird durch den neuen String ersetzt.
- Erzeugen einer Kopie, wobei der Speicher für den Zielstring als Parameter übergeben wird:

```
void upCaseCopy(char dest[], char src[]) {
    // TODO: Implementierung
}
```

Aufgabe 9.3 *Lexikographischer Stringvergleich*

Vergleichen Sie zwei Strings bezüglich ihrer Reihenfolge im Wörterbuch. Beachten Sie dabei, dass bei einem Zeichenvergleich alle Kleinbuchstaben kleiner sind als die Großbuchstaben. Verwenden Sie dazu die vorige Funktion.

Aufgabe 9.4 *Strings vertauschen*

Schreiben Sie eine Funktion, die den Inhalt zweier Stringvariablen vertauscht. Wir gehen davon aus, dass die benutzten `char`-Felder groß genug sind, um beide Strings aufzunehmen.

- *Selection Sort*
- *Insertion Sort*
- *Schnelle Sortierverfahren*
- *Sortieren von Stringfeldern*
- *Sortieren von Structs*
- *Aufgaben*

Wir haben bereits bei der binären Suche gesehen, dass sich bestimmte Aufgaben vereinfachen oder beschleunigen lassen, wenn unsere Daten sortiert vorliegen. Auch für die reine Ausgabe oder manuelle Interpretation der Daten ist eine Sortierung häufig hilfreich.

Unsere einzige bisher bekannte Datenstruktur zur Verwaltung von einer größeren Anzahl von Werten ist das Feld, das wir nun sortieren wollen.

Voraussetzung für die Sortierung ist das Vorliegen einer *Ordnungsrelation*, wir müssen also für zwei Werte entscheiden können, in welcher Reihenfolge sie anzuordnen sind.

Um die Ergebnisse unserer Algorithmen schnell auswerten zu können, verwenden wir einfach ein Zeichenfeld, das wir mit dem String:

```
bittesortieremich
```

initialisieren:

```
char s[] = "bittesortieremich";  
int len = strlen(s); // in string.h deklariert
```

Wir wissen, dass die einzelnen Zeichen den Datentyp `char` haben, der ein Zahlentyp ist. Die Zeichen werden also bzgl. ihres *ASCII*-Codes sortiert.

Aus *AuP* kennen wir das *Bubblesort*-Verfahren, das wir hier nur noch einmal zum Vergleich anführen, aber nicht weiter betrachten wollen.

Quellcode 10.1: bubblesort.c

```

/* Bubblesort zum Vergleich.
 */
#include <stdio.h>
#include <stdbool.h>
#include <string.h>

/* Hilfsfunktion: zwei Feldelemente vertauschen.
 */
void exch(char s[], int idx1, int idx2) {
    char h = s[idx1];
    s[idx1] = s[idx2];
    s[idx2] = h;
}

void bubbleSort(char s[], int len) {
    bool isChanged = true;
    // wiederhole, solange es noch Vertauschungen gibt
    while (isChanged) {
        isChanged = false;
        for (int i = 0; i < len - 1; i++) {
            if (s[i] > s[i + 1]) {
                exch(s, i, i + 1);
                isChanged = true;
            }
        }
    }
}

int main() {
    char s[] = "bittesortieremich";
    int len = strlen(s);
    printf("Original: %s\n", s);
    bubbleSort(s, len);
    printf("Sortiert: %s\n", s);
}
    
```

Wir erhalten die Ausgabe:

```

Original: bittesortieremich
Sortiert: bceeehiiimorrsttt
    
```

Der Aufwand des Verfahrens ist im *worst case* $O(n^2)$: Wir benötigen n Schleifendurchläufe (Warum?) mit jeweils höchstens n Tests und eventuellen Vertauschungen. Dieser Fall tritt auf, wenn das Feld gerade in absteigend sortiert ist.

10.1 Selection Sort

Die Idee dieses Verfahrens ist folgende: Wir suchen uns den kleinsten Wertes des Feldes heraus und tauschen ihn auf die erste Position. Anschließend betrachten wir das Restfeld ohne das erste Element und wiederholen das Verfahren: wir suchen hier wieder das kleinste Element und tauschen es auf Position 2. Das setzen wir fort, bis unser Restfeld die Größe 1 hat.

Quellcode 10.2: selsort.c

```

/* Selection Sort
 */
#include <stdio.h>
#include <stdbool.h>
#include <string.h>

/* Hilfsfunktion: zwei Feldelemente vertauschen.
 */
void exch(char s[], int idx1, int idx2) {
    char h = s[idx1];
    s[idx1] = s[idx2];
    s[idx2] = h;
}

void selSort(char s[], int len) {
    for (int i = 0; i < len - 1; i++) {
        // Minimum ab i suchen
        int iMin = i;
        // sinnvoll für große Felder: char sMin = s[i];
        for (int k = i + 1; k < len; k++) {
            if (s[k] < s[iMin]) { // besser s[k] < sMin
                iMin = k;      // und: sMin = s[k]
            }
        }
        // Minimum nicht an Position i: tauschen
        if (iMin != i) {
            exch(s, iMin, i);
        }
    }
}

int main() {
    char s[] = "bittesortieremich";
    int len = strlen(s);
    printf("Original: %s\n", s);
    selSort(s, len);
    printf("Sortiert: %s\n", s);
}

```

Wir haben auch hier wieder eine Worst-Case-Komplexität von $O(n^2)$, wobei das allerdings nur für die Vergleiche zutrifft. Allein für die Vertauschungen erhalten wir eine Komplexität von $O(n)$. Dieses Verfahren ist also dann in Erwägungen zu ziehen, wenn unsere Felder auf Speichermedien liegen, die zwar schnell gelesen werden können, bei denen das Schreiben aber lange dauert. Sofern unsere Felder komplett im Hauptspeicher liegen, ist der Aufwand für beide Operationen aber vergleichbar. Meist ist *Selection Sort* etwas schneller als *Bubble Sort*.

Bemerkung: Wenn wir sehr große Felder mit diesem Verfahren sortieren wollen, stellen wir bei dieser Implementierung allerdings ein deutlich schlechteres Laufzeitverhalten fest. Dies ist bedingt durch den Zugriff auf weit voneinander entfernt liegende Feldelemente, die nicht gleichzeitig in den Cache geladen werden. Durch Zwischenspeichern des aktuellen Minimums zusätzlich zum Index dieses Elements lässt sich dieser Effekt eliminieren.

10.2 Insertion Sort

Bei diesem Sortierverfahren nehmen wir das erste Element her (was automatisch sortiert ist) und sortieren nun das zweite Element an die richtige Stelle vor oder hinter das erste Element. Wir setzen mit dem dritten Element fort, das wir an der richtigen Stelle in die ersten beiden Elemente einsortieren. Schritt für Schritt wird ein weiteres Element in die bereits sortierten Elemente eingefügt, bis jedes Element einsortiert ist.

Praktisch gehen wir so vor, dass wir das neu einzusortierende Element in einer Hilfsvariable speichern und alle links von ihm liegenden Elemente eine Position nach rechts rücken, bis sich die entstandene Lücke an der richtigen Stelle befindet, in der das neue Element aus der Hilfsvariable eingefügt wird.

Quellcode 10.3: inssort.c

```
/* Insertion Sort
 */
#include <stdio.h>
#include <stdbool.h>
#include <string.h>

void insSort(char s[], int len) {
    for (int i = 1; i < len; i++) {
        // Schaffe Lücke an Position i
        char h = s[i];
        // Schiebe alle Elemente > h nach rechts
        int k = i;
        while (k > 0 && s[k-1] > h) {
            s[k] = s[k-1];
            k--;
        }
        // Schiebe h in die Lücke
        s[k] = h;
    }
}

int main() {
    char s[] = "bittesortieremich";
    int len = strlen(s);
    printf("Original: %s\n", s);
    insSort(s, len);
    printf("Sortiert: %s\n", s);
}
```

Die Worst-Case-Komplexität ist wieder $O(n^2)$. In der Praxis ist jedoch *Insertion Sort* meist das schnellste dieser 3 Verfahren. Ist das Feld nämlich schon teilweise sortiert, müssen die meisten Elemente nicht weit wandern und die innere Schleife wird nur wenige Male durchlaufen. Diese Idee lässt sich weiter ausbauen, man erhält daraus z. B. das *Shell-Sort-Verfahren* mit einer deutlich besseren Worst-Case-Komplexität (siehe [Sed14]).

10.3 Schnelle Sortierverfahren

Die bisherigen Sortierverfahren sind geeignet, um einige Tausend bis Zehntausend Elemente in akzeptabler Zeit zu sortieren. Zum Sortieren viel größerer Datenbestände von einigen Millionen oder noch mehr Elementen scheitern sie jedoch an der zu großen Laufzeit. Glücklicherweise gibt es raffiniertere Verfahren, die die Worst-Case-Komplexität auf $O(n \log n)$ reduzieren. Viele beruhen auf dem rekursiven *Divide-and-Conquer*-Ansatz: Wir zerlegen das Gesamtfeld in zwei Teilfelder, sortieren diese und fügen

die beiden sortierten Felder zu einem gemeinsamen Feld zusammen. Für das Sortieren der Teilfelder gehen wir analog vor. Unser Algorithmus hat also folgende Grundstruktur:

```
sortiere(feld) {
    if größe(feld) == 1 {
        // Feld ist automatisch sortiert
        return
    }
    feld1, feld2 = zerlege(feld)
    sortiere(feld1)
    sortiere(feld2)
    feld = vereinigeSortiert(feld1, feld2)
    return
}
```

Die eigentliche Arbeit dieser Verfahren liegt in der geschickten Zerlegung des Feldes in Teilfelder und das Zusammenfügen der sortierten Felder.

Im nächsten Semester wird das *Quicksort*-Verfahren als praktisch schnellster Vertreter dieser Klasse genauer untersucht. Hier betrachten wir das leichter zu verstehende *Mergesort*-Verfahren, bei dem die Zerlegung in Teilfelder einfach durch Halbierung des Feldes in der Mitte erfolgt. Dafür ist das Zusammenfügen etwas aufwendiger:

Uns liegen zwei sortierte Teilfelder vor, die zu einem sortierten Feld zusammengeführt werden müssen. Dazu setzen wir je einen Zeiger (praktisch einen Index) auf jeden Feldanfang. Von diesen beiden aktiven Elementen nehmen wir das kleinere und schieben den entsprechenden Zeiger eine Position weiter. Nun wiederholen wir wieder den Vergleich der beiden Kopfelemente, bis der Reihe nach alle Elemente in das neue Feld übernommen sind. In einer der beiden Listen sind nun noch einige (sortierte) Elemente übrig, die einfach an das neue Feld angehängt werden können.

Quellcode 10.4: mergesort.c

```
/* Merge Sort
 */
#include <stdio.h>
#include <stdbool.h>
#include <string.h>
#include <assert.h>

/* Sortierte Teilfelder left...m-1 und m...right-1 sortiert zusammenfügen.
 */
void joinParts(char s[], int left, int m, int right) {
    // Hilfsfeld erzeugen 0...right-1
    char h[right];
    // Umkopieren, 0...left-1 ungenutzt
    for (int i = left; i < right; i++) {
        h[i] = s[i];
    }
    int iLeft = left;
    int iRight = m;
    int i = left;
    while (iLeft < m && iRight < right) {
        // Wähle jeweils kleineres Kopfelement als nächstes.
        if (h[iLeft] < h[iRight]) {
            s[i] = h[iLeft];
            iLeft++;
        } else {
            s[i] = h[iRight];

```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```

        iRight++;
    }
    i++;
}
// Eins der beiden Teilfelder enthält noch Elemente, noch anhängen
while (iLeft < m) {
    s[i] = h[iLeft];
    iLeft++;
    i++;
}
while (iRight < right) {
    s[i] = h[iRight];
    iRight++;
    i++;
}
assert(i == right);
}

/* Sortiere Teilfeld left ... right-1.
 */
void mergeSortRec(char s[], int left, int right) {
    if (right-left < 2) return;
    int m = (left + right)/2;
    mergeSortRec(s, left, m);
    mergeSortRec(s, m, right);
    joinParts(s, left, m, right);
}

void mergeSort(char s[], int len) {
    mergeSortRec(s, 0, len);
}

int main() {
    char s[] = "bittesortieremich";
    int len = strlen(s);
    printf("Original: %s\n", s);
    mergeSort(s, len);
    printf("Sortiert: %s\n", s);
}

```

Schrittweises Zusammenfügen:

```

Original: bittesortieremich
Teilfelder:
* 0-1: b
* 1-2: i
Zusammengefügt:
* 0-2: bi
Teilfelder:
* 2-3: t
* 3-4: t
Zusammengefügt:
* 2-4: tt
Teilfelder:
* 0-2: bi
* 2-4: tt

```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```

Zusammengefügt:
* 0-4: bitt
  Teilfelder:
  * 4-5: e
  * 5-6: s
  Zusammengefügt:
  * 4-6: es
  Teilfelder:
  * 6-7: o
  * 7-8: r
  Zusammengefügt:
  * 6-8: or
  Teilfelder:
  * 4-6: es
  * 6-8: or
  Zusammengefügt:
  * 4-8: eors
Teilfelder:
* 0-4: bitt
* 4-8: eors
Zusammengefügt:
* 0-8: beiorstt
  Teilfelder:
  * 8-9: t
  * 9-10: i
  Zusammengefügt:
  * 8-10: it
  Teilfelder:
  * 10-11: e
  * 11-12: r
  Zusammengefügt:
  * 10-12: er
  Teilfelder:
  * 8-10: it
  * 10-12: er
  Zusammengefügt:
  * 8-12: eirt
  Teilfelder:
  * 12-13: e
  * 13-14: m
  Zusammengefügt:
  * 12-14: em
  Teilfelder:
  * 15-16: c
  * 16-17: h
  Zusammengefügt:
  * 15-17: ch
  Teilfelder:
  * 14-15: i
  * 15-17: ch
  Zusammengefügt:
  * 14-17: chi
Teilfelder:
* 12-14: em
* 14-17: chi
Zusammengefügt:

```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
* 12-17: cehim
Teilfelder:
* 8-12: eirt
* 12-17: cehim
Zusammengefügt:
* 8-17: ceehiimrt
Teilfelder:
* 0-8: beiorstt
* 8-17: ceehiimrt
Zusammengefügt:
* 0-17: bceeehiimorrsttt
Sortiert: bceeehiimorrsttt
```

Download der Version mit Debug-Ausgabe: `mergesort_debug.c`.

Bemerkung: Das Zusammenfügen kann geschickter erfolgen, indem das zweite Teilfeld in *umgekehrter* Reihenfolge in das Hilfsfeld umkopiert wird. Dies ist in [Sed14] genauer erläutert.

Die folgende Tabelle zeigt die Laufzeit verschiedener Sortierverfahren für verschiedene Feldgrößen im Vergleich. Statt Mergesort wurde hier Heapsort verwendet, das ebenfalls die Komplexität $O(n \log n)$ hat. Die Zeiten in der Tabelle sind Mikrosekunden.

Size	Bubble	Heap	Insertion	Selection
20	8	7	3	6
50	44	17	10	18
100	183	38	28	52
200	671	85	110	172
500	2236	95	271	384
1000	5929	202	988	1473
2000	23965	469	4146	5748
5000	149112	1376	25189	35378
10000	598194	2951	99712	140653
15000	1341377	4771	223094	315725
20000	2446892	6387	403151	560503
25000	3986226	7843	629097	875318
35000	7847842	11463	1216362	1714051
50000	16702618	17746	2491231	3495803

Das Sortieren von 50000 Zahlen dauert also auf dem Testsystem mit *Bubblesort* bereits 17 Sekunden, mit *Insertion Sort* dagegen nur 2,5 Sekunden, obwohl beide Verfahren die gleichen Worst-Case-Komplexität haben. Mit *Heapsort* dauert das Sortieren des gleichen Feldes dagegen nur 18 Millisekunden.

Eine graphische Darstellung zeigt deutlich, wie langsam Bubblesort im Vergleich zu Selection bzw. Insertion Sort ist, und wie groß die Überlegenheit eines schnellen Verfahrens bei größeren Datenmengen ist.

10.4 Sortieren von Stringfeldern

Beim Sortieren von Stringfeldern haben wir mehrere Besonderheiten zu beachten:

- Der Vergleich zweier Strings ist nicht mit `<` oder ähnlichen Operatoren möglich. Wir benötigen eine Funktion, die die Strings von links nach rechts zeichenweise vergleicht. Die Standardbibliothek stellt dafür die Funktion `strcmp()` zur Verfügung. Statt

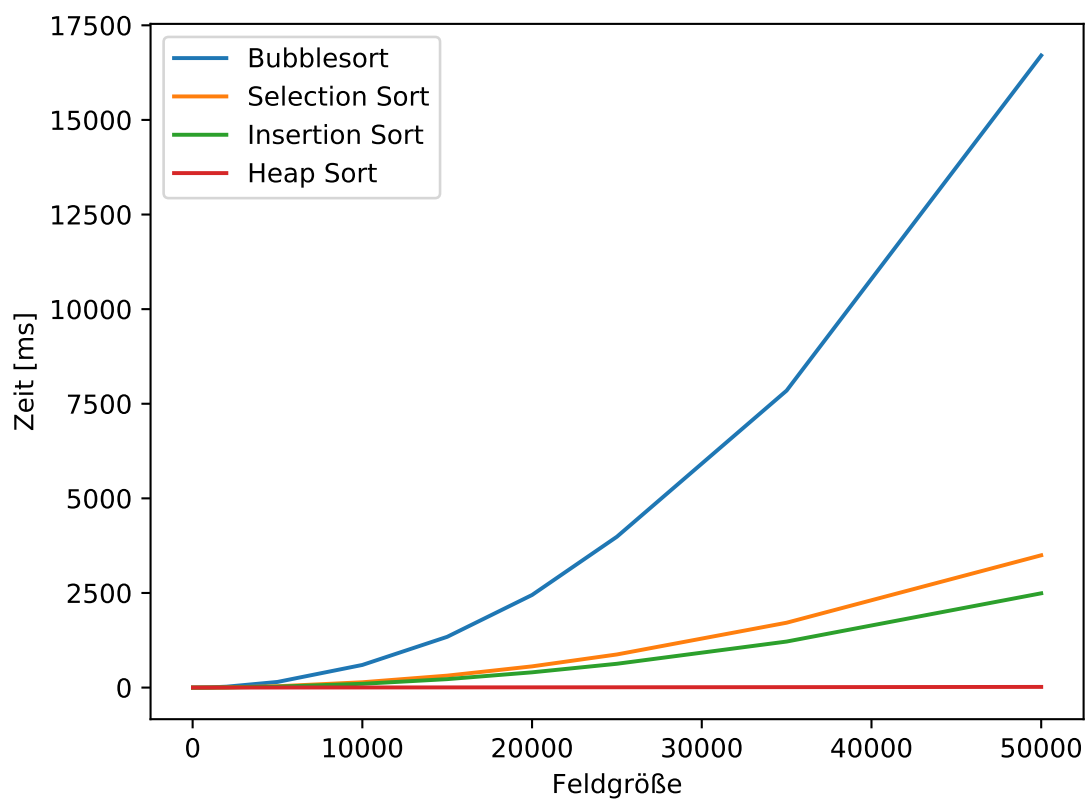


Abb. 10.1: Laufzeit verschiedener Sortiervverfahren im Vergleich.

```
if (a < b)
```

schreiben wir nun

```
if (strcmp(a, b) < 0)
```

- Der zeichenweise Vergleich erfolgt bezüglich des *ASCII*-Codes der Zeichen. Das bedeutet, dass das kleine *a* als kleiner als das große *Z* betrachtet wird.
- Unser Stringfeld ist ein zweidimensionales Feld (sofern es keine Verweise auf konstante Strings enthält). Der Vertausch zweier Strings bedeutet das komplette zeichenweise Vertauschen zweier Zeilen unseres Feldes.

Wir müssen uns für die Teilaufgaben Vergleich und Vertauschen passende Hilfsfunktionen schreiben, dann können unsere üblichen Sortierverfahren zum Einsatz kommen.

10.5 Sortieren von Structs

Im Gegensatz zu Stringfeldern können wir in Feldern von Structs die Elemente direkt vertauschen. Allerdings ist ein Vergleich zweier Struct-Variablen mit *<* und ähnlichen Operatoren ebenfalls nicht möglich und auch wenig sinnvoll, denn die Ordnung wird durch einzelne Komponenten bestimmt bzw. muss wie bei unseren Struct für Brüche erst berechnet werden.

Wir müssen also auch hier eine Vergleichsfunktion passend definieren.

Betrachten wir wieder unser Beispiel *Fraction* (ohne Gleitpunktdarstellung):

```
struct Fraction {
    int nom;
    int denom;
};
```

Unsere Vergleichsfunktion bringt beide Brüche auf den Hauptnenner und vergleicht anschließend die Zähler:

```
/* Vergleich zweier Brüche.
 * :return: res < 0 , wenn a < b, res == 0, wenn a == b, res > 0 sonst
 */
int cmpFractions(struct Fraction a, struct Fraction b) {
    // auf Hauptnenner a.denom * b.denom bringen
    // benötigen nur die Zähler
    a.nom *= b.denom;
    b.nom *= a.denom;
    return a.nom - b.nom;
}
```

Komplettbeispiel: Die Brüche werden als Paare ganzer Zahlen gelesen, die Eingabe wird mit dem Paar (0, 0) abgeschlossen. Zur besseren Kontrolle wird in der Ausgabe die Gleitpunktdarstellung der Brüche berechnet.

Quellcode 10.5: *fracsort.c*: Sortieren von Brüchen

```
/* Sortieren von Brüchen
 * Eingabe aus Datei (Paare Zähler-Nenner), Abschluss (0, 0)
 */
#include <stdio.h>
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
enum {MAX_FRACS = 1000};

struct Fraction {
    int nom;
    int denom;
};

int readInt() {
    int res;
    scanf("%d", &res);
    return res;
}

/* Einlesen der Brüche als Paare bis maxFracS bzw. Paar 0 0
 * :return: Anzahl der gelesenen Brüche.
 */
int readFractions(struct Fraction fractions[], int maxFracS) {
    int nom = readInt();
    int denom = readInt();
    int count = 0;
    while (denom != 0 && count < maxFracS) {
        // Normalisieren
        if (denom < 0) {
            nom = -nom;
            denom = -denom;
        }
        fractions[count].nom = nom;
        fractions[count].denom = denom;
        nom = readInt();
        denom = readInt();
        count++;
    }
    return count;
}

void showFraction(struct Fraction f) {
    printf("%d/%d (%.2f)\n", f.nom, f.denom, ((double)f.nom)/f.denom);
}

void showFractions(struct Fraction fractions[], int count) {
    for (int i = 0; i < count; i++) {
        showFraction(fractions[i]);
    }
}

/* Vergleich zweier Brüche.
 */
int cmpFractions(struct Fraction a, struct Fraction b) {
    // auf Hauptnenner a.denom * b.denom bringen
    // benötigen nur die Zähler
    a.nom *= b.denom;
    b.nom *= a.denom;
    return a.nom - b.nom;
}

/* Vertauschen zweier Brüche im Feld.
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```

*/
void exchangeFractions(struct Fraction fractions[], int i, int k) {
    struct Fraction help = fractions[i];
    fractions[i] = fractions[k];
    fractions[k] = help;
}

/* SelectionSort für Brüche.
*/
void sortFractions(struct Fraction fractions[], int count) {
    for (int i = 0; i < count - 1; i++) {
        int iMin = i;
        for (int k = i + 1; k < count; k++) {
            if (cmpFractions(fractions[k], fractions[iMin]) < 0) {
                iMin = k;
            }
        }
        if (iMin != i) {
            exchangeFractions(fractions, i, iMin);
        }
    }
}

int main() {
    struct Fraction fractions[MAX_FRACS];
    int count = 0;

    count = readFractions(fractions, MAX_FRACS);
    printf("Unsortiert:\n");
    showFractions(fractions, count);
    sortFractions(fractions, count);
    printf("\nSortiert:\n");
    showFractions(fractions, count);
}
    
```

Download: fracsort.c. Datendatei: fractions.dat. Ergebnis für diese Datei:

```

Unsortiert:
1/3 (0.33)
2/5 (0.40)
1/5 (0.20)
-3/5 (-0.60)
7/8 (0.88)
9/2 (4.50)

Sortiert:
-3/5 (-0.60)
1/5 (0.20)
1/3 (0.33)
2/5 (0.40)
7/8 (0.88)
9/2 (4.50)
    
```

10.6 Aufgaben

Aufgabe 10.1 *Sortieren eines Zahlenfeldes*

Lesen Sie eine Folge von maximal 1000 positiven Zahlen, die mit 0 terminiert wird, von der Konsole in ein Feld ein und sortieren Sie dieses. Schreiben Sie dazu geeignete Tests, die sicherstellen, dass das sortierte Feld tatsächlich die gleichen Werte wie das Ausgangsfeld enthält.

Aufgabe 10.2 *Absteigendes Sortieren*

Implementieren Sie das absteigende Sortieren der Zahlen. Schreiben Sie anschließend eine Sortierfunktion, die die Sortierrichtung (1 für aufsteigend, -1 für absteigend) als Parameter erhält.

Aufgabe 10.3 *Sortieren eines Stringfeldes*

Implementieren Sie die Stringsortierung. Lesen Sie dazu maximal 1000 Strings der maximalen Länge 50 (terminiert mit dem Endekennzeichen ###) in ein Feld ein und sortieren Sie dieses.

Aufgabe 10.4 *Sortieren von Structs*

Entwerfen Sie einen Struct **Person** mit den Komponenten Name und Alter. Schreiben Sie Funktionen, die ein Personenfeld:

- nach dem Namen
- nach dem Alter, und bei gleichem Alter nach dem Namen

sortiert. Zeigen Sie die jeweiligen Ergebnisse an.

- *Konzept*
- *Modellierung als Datenstruktur*
- *Bipartite Graphen*
- *Aufgaben*

11.1 Konzept

Sehr oft müssen wir Beziehungen zwischen Objekten modellieren, z.B. bieten verschiedene Firmen unterschiedliche Produkte zu unterschiedlichen Konditionen an, bestimmte Personen sind miteinander befreundet, hassen sich oder kennen sich gar nicht, Städte sind durch verschiedene Transportwege mit verschiedenen Längen, Kosten, Fahrzeiten miteinander verbunden. Solche paarweisen Beziehungen zwischen Objekten lassen sich sehr gut mit *Graphen* modellieren.

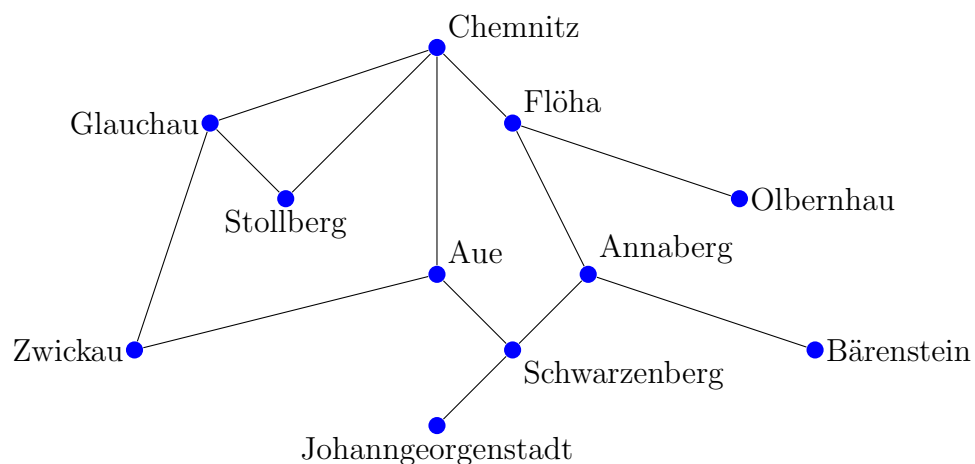


Abb. 11.1: Teil des erzgebirgischen Eisenbahnnetzes.
Den Kanten (Verbindungen) können weitere Eigenschaften wie Entfernung oder Fahrzeit zugeordnet werden.

Wir nennen die Objekte *Knoten* (engl. *vertices*, Singular *vertex*) und die Beziehungen/Verbindungen *Kanten* (engl. *edges*). Ordnen wir den Kanten noch Werte zu, werden diese als *Gewichte* (engl. *weights*) bezeichnet da es oft um numerische Werte handelt, die in irgendeiner Form Kosten beschreiben.

Oft haben wir auch zwei unterschiedliche Objektklassen wie Produzenten und Produkte, und wir betrachten nur Beziehungen zwischen Objekten verschiedener Klassen, z. B. die Information, welcher Produzent welche Produkte und vielleicht auch in welcher Menge und zu welchem Preis fertigt. Im Beispiel betrachten wir die Wochentage und die an diesen Tagen zu besuchenden TU-Gebäude.

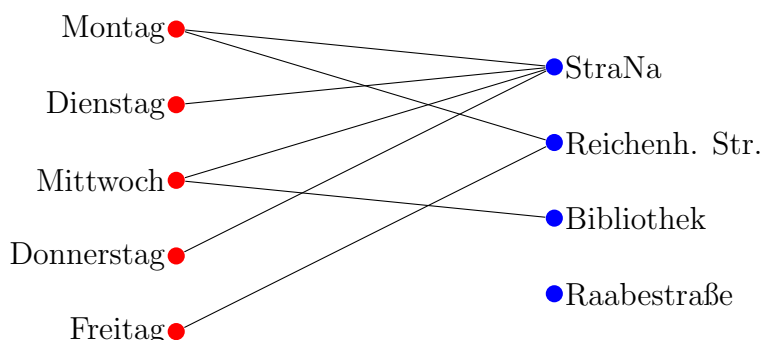


Abb. 11.2: Beziehung zwischen Wochentagen und den an diesen Tagen besuchten Untereinheiten. Diesen Beziehungen könnte noch die Dauer oder der Grund des Aufenthaltes am jeweiligen Tag und Ort zugeordnet werden.

Diese Beziehungen können wir ebenfalls als Graph modellieren. Er hat die besondere Eigenschaft, dass nur Kanten zwischen zwei Knoten verschiedener Klassen existieren. Diese Graphen nennt man *bipartit* (zweiteilig) oder auch 2-färbbar, da hier zwei Farben genügen um den Graph so zu färben, dass Kanten nur Knoten unterschiedlicher Farbe verbinden (in der Abbildung rot und blau dargestellt).

11.2 Modellierung als Datenstruktur

Unsere Knotenmenge können wir einfach als ein Feld modellieren, das jeden Knoten als Element enthält. Oft hat das Objekt, das einen Knoten repräsentiert, mehrere Eigenschaften. Es bietet sich also ein Feld von Structs an.

Beispiel: Ort.

Quellcode 11.1: Modellierung der Ortsknoten

```
struct Place {
    char name[MAX_LEN];
    double lon; // Längengrad
    double lat; // Breitengrad
};

struct Place nodes[MAX_NODES];
int numPlaces = 0; // tatsächliche Knotenzahl
```

Bei bipartiten Graphen haben wir zwei Objektmengen mit unterschiedlichen Eigenschaften, wir verwenden deshalb für jede Objektmenge ein eigenes Feld.

Um die Kanten, also die Beziehungen zwischen Objekten, zu modellieren, gibt es viele Möglichkeiten. Wir beschränken uns hier auf die einfache Variante der *Adjazenzmatrix*. Dazu nummerieren wir die Knoten einfach durch (= Indices unseres Knotenfeldes) und erstellen ein zweidimensionales `bool`-Feld. An die Position `[i][k]` schreiben wir `true`, wenn eine Kante zwischen den Knoten `i` und `k` existiert, und sonst `false`.

Quellcode 11.2: Modellierung der Kanten

```
// Feld wird bis zur Größe (numPlaces - 1) * (numPlaces - 1) genutzt.
bool adj[MAX_NODES][MAX_NODES] = {false};
```

Diese Matrix ist normalerweise symmetrisch, also `adj[i][k] == adj[k][i]` und von einem Knoten zu sich selbst existiert keine Kante: `adj[i][i] == 0`.

Benötigen wir Gewichte für die Kanten, verwenden wir statt `bool` den Typ `int` für die Feldelemente der Adjazenzmatrix. Wenn wir davon ausgehen können, dass alle Gewichte positiv sind, können wir nun den Wert -1 verwenden, um zu kennzeichnen, dass es zwischen zwei Knoten keine Kante gibt.

Bemerkung: Sofern möglich, sollten Gewichte immer positiv modelliert werden, da viele Optimierungsalgorithmen auf Graphen nur dann funktionieren.

Beispiel: Adjazenzmatrix für den Erzgebirgsbahngraphen.

Quellcode 11.3: Adjazenzmatrix für Bahngraphen.

Ort		Ann.	Aue	Bär.	Ch.	Flö.	Gla.	Joh.	Olb.	Schw.	Sto.	Zwi.
Idx		0	1	2	3	4	5	6	7	8	9	10
Ann	0	-	-	X	-	X	-	-	-	X	-	-
Aue	1	-	-	-	X	-	-	-	-	X	-	X
Bär	2	X	-	-	-	-	-	-	-	-	-	-
Ch	3	-	X	-	-	X	X	-	-	-	X	-
Flö	4	X	-	-	X	-	-	-	X	-	-	-
Gla	5	-	-	-	X	-	-	-	-	-	X	X
Joh	6	-	-	-	-	-	-	-	-	X	-	-
Olb	7	-	-	-	-	X	-	-	-	-	-	-
Sch	8	X	X	-	-	-	-	X	-	-	-	-
Sto	9	-	-	-	X	-	X	-	-	-	-	-
Zw	10	-	X	-	-	-	X	-	-	-	-	-

Die Ortsbezeichnungen sind NICHT Bestandteil der Matrix und dienen nur zur besseren Orientierung.

Um den Graphen aufzubauen, benötigen wir zunächst die Knotenmenge. Wir ordnen jedem Knoten eine Nummer zu, praktisch ist das meist der Index des Objekts im Knotenfeld. Wir erzeugen zunächst eine Adjazenzmatrix der benötigten Größe und setzen jeden Eintrag auf `false`. Nun werden die Kanten durch Angabe der Knoten hinzugefügt:

```
void addEdge(int size, bool adj[size][size], int n1, int n2) {
    adj[n1][n2] = adj[n2][n1] = true;
}
```

Beispiel: Einlesen des Bahngraphen. Jeder Ort besteht aus Name und den geographischen Koordinaten. Wir erzeugen eine Datendatei mit folgendem Aufbau:

- Ortsliste, jeder Ort mit Koordinaten in einer Zeile, durch ; getrennt.
- Abschluss Ortsliste mit dem Name ### und beliebigen Koordinaten
- Liste der Kanten als: Ortsindex1, Ortsindex2, durch Leerzeichen getrennt
- Abschluss der Kantenliste mit den Werten -1, -1

Die Zahl der Orte soll auf 100 beschränkt, die Länge der Ortsnamen nicht größer als 30 sein. Für unser Bahnnetz-Beispiel erhalten wir folgende Eingabedatei:

Quellcode 11.4: bahngraph.dat: Daten des Graphen

```
Annaberg;13.01;50.58
Aue;11.70;50.59
Baerenstein;13.03;50.51
Chemnitz;12.91;50.87
Floeha;13.08;50.85
Glauchau;12.54;50.82
Johanngeorgenstadt;12,71;50.43
Olbernhau;13,35;50.66
Schwarzenberg;12.78;50.54
Stollberg;12.78;50.71
Zwickau;12.49;50.72
###;0;0
0 2
0 4
0 8
1 3
1 8
1 10
3 4
3 5
3 9
4 7
5 9
5 10
6 8
-1 -1
```

Download: [bahngraph.dat](#).

Um den Graph einzulesen, erstellen wir zunächst die Felder für die Ortsstruktur mit 100 sowie die Adjazenzmatrix für die Entfernungen mit 100×100 `int`-Einträgen. Wir lesen nun die Orte Zeile für Zeile ein (maximale Zeilenlänge: maximale Ortslänge + 2-mal Zahlenlänge + 2 Trennzeichen) und zerlegen diese am Semikolon in ihre drei Bestandteile. Ist das Endekennzeichen nicht erreicht, werden die Daten in das Ortsfeld übernommen. Anschließend lesen wir die Kanten als Knotenpaare einfach durch jeweils 2 `readInt()`-Aufrufe und tragen am passenden Index sowie an der gespiegelten Position ein `true` ein.

Für die Anzeige der Adjazenzmatrix erzeugen wir in einer Schleife über die Spalten eine Kopfzeile mit den Ortsindizes, dann folgt eine Schleife über die Zeilen, die zunächst den Zeilenindex und anschließend in einer inneren Schleife über die Spalten die Matrixelemente dieser Zeile ausgibt.

Wir geben die Verbindungen noch einmal getrennt aus, indem wir das obere Dreieck der Adjazenzmatrix ohne die Diagonale durchlaufen und bei jedem `true`-Wert den Zeilen- und Spaltenindex benutzen, um die zugehörigen Orte der Endknoten der Kante auszuwählen und anzuzeigen.

Quellcode 11.5: bahngraph.c: Einlesen und Anzeige des Bahngraphen

```
/* Bahngraph.
 */

#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <string.h>

enum {MAX_NODES = 100, MAX_STRLEN = 30, MAX_LINE = 60};
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```

struct Place {
    char name[MAX_STRLEN + 1];
    double lon;
    double lat;
};

int readInt() {
    int res;
    scanf("%d", &res);
    return res;
}

/* Eingabezeile Knoten in Struct umwandeln.
 * :param line: Eingabezeile der Form Ort;Länge;Breite
 * :return: Place-Objekt
 *
 * Die Entfernung des \n am Ende von line ist nicht erforderlich.
 */
struct Place line2place(char line[]) {
    struct Place result;
    char *part = strtok(line, ";");
    strncpy(result.name, part, MAX_STRLEN);
    part = strtok(NULL, ";");
    result.lon = atof(part);
    part = strtok(NULL, ";");
    result.lat = atof(part);
    return result;
}

/* Graph einlesen.
 * :return: Anzahl Knoten
 *
 * Liest die Orte bis ###;0;0
 * Anschließend die Kanten in der Form index1 index2 bis -1 -1
 */
int readGraph(struct Place places[], bool adj[MAX_NODES][MAX_NODES]) {
    int numPlaces = 0;
    // Knoten einlesen
    char line[MAX_LINE + 1];
    fgets(line, MAX_LINE, stdin);
    struct Place place = line2place(line);
    while (strcmp(place.name, "###") != 0) {
        places[numPlaces] = place;
        numPlaces++;
        fgets(line, MAX_LINE, stdin);
        place = line2place(line);
    }
    // Kanten lesen
    int n1 = readInt();
    int n2 = readInt();
    while (n1 >= 0) {
        adj[n1][n2] = adj[n2][n1] = true;
        n1 = readInt();
        n2 = readInt();
    }
}
    
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```

    return numPlaces;
}

void showPlaces(int numPlaces, struct Place places[]) {
    for (int i = 0; i < numPlaces; i++) {
        printf("%-30s [%7.2f, %7.2f]\n", places[i].name, places[i].lon,
            places[i].lat);
    }
}

void showAdj (int numPlaces, bool adj[MAX_NODES][MAX_NODES]) {
    printf(" |");
    for (int to = 0; to < numPlaces; to++) {
        printf("%2d ", to);
    }
    printf("\n");
    printf("--|");
    for (int to = 0; to < numPlaces; to++) {
        printf("----");
    }
    printf("\n");
    for (int from = 0; from < numPlaces; from++) {
        printf("%2d|", from);
        for (int to = 0; to < numPlaces; to++) {
            if (adj[from][to]) {
                printf(" X ");
            } else {
                printf(" - ");
            }
        }
        printf("\n");
    }
}

/* Anzeige des Ortspaares (from, to).
*/
void showEdge(struct Place places[], int from, int to) {
    printf("%s -- %s\n", places[from].name, places[to].name);
}

/* Anzeige aller Kanten als Ortspaar.
*/
void showEdges(int numPlaces, struct Place places[], bool adj[][MAX_NODES]) {
    for (int from = 0; from < numPlaces; from++) {
        for (int to = from + 1; to < numPlaces; to++) {
            if (adj[from][to]) {
                showEdge(places, from, to);
            }
        }
    }
}

int main() {
    struct Place places[MAX_NODES];
    int numPlaces = 0;
    bool adj[MAX_NODES][MAX_NODES] = {false};

```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```

numPlaces = readGraph(places, adj);
printf("Orte:\n\n");
showPlaces(numPlaces, places);
printf("\nAdjanzenzmatrix:\n\n");
showAdj(numPlaces, adj);
printf("\nVerbindungen:\n\n");
showEdges(numPlaces, places, adj);
}
    
```

Ergebnis für unsere Eingabe:

Orte:

Annaberg	[13.01,	50.58]
Aue	[11.70,	50.59]
Baerenstein	[13.03,	50.51]
Chemnitz	[12.91,	50.87]
Floeha	[13.08,	50.85]
Glauchau	[12.00,	50.82]
Johanngeorgenstadt	[12.00,	50.43]
Olbernhau	[13.00,	50.66]
Schwarzenberg	[12.78,	50.54]
Stollberg	[12.78,	50.71]
Zwickau	[12.49,	50.72]

Adjanzenzmatrix:

	0	1	2	3	4	5	6	7	8	9	10
0	-	-	X	-	X	-	-	-	X	-	-
1	-	-	-	X	-	-	-	-	X	-	X
2	X	-	-	-	-	-	-	-	-	-	-
3	-	X	-	-	X	X	-	-	-	X	-
4	X	-	-	X	-	-	-	X	-	-	-
5	-	-	-	X	-	-	-	-	-	X	X
6	-	-	-	-	-	-	-	-	X	-	-
7	-	-	-	-	X	-	-	-	-	-	-
8	X	X	-	-	-	-	X	-	-	-	-
9	-	-	-	X	-	X	-	-	-	-	-
10	-	X	-	-	-	X	-	-	-	-	-

Verbindungen:

```

Annaberg -- Baerenstein
Annaberg -- Floeha
Annaberg -- Schwarzenberg
Aue -- Chemnitz
Aue -- Schwarzenberg
Aue -- Zwickau
Chemnitz -- Floeha
Chemnitz -- Glauchau
Chemnitz -- Stollberg
Floeha -- Olbernhau
Glauchau -- Stollberg
Glauchau -- Zwickau
Johanngeorgenstadt -- Schwarzenberg
    
```

11.3 Bipartite Graphen

Für bipartite Graphen können wir die Adjazenzmatrix etwas vereinfachen, da es innerhalb der beiden Knotenmengen keine Kanten gibt. Wir verwenden die Zeilen der Matrix für die Knoten der ersten und die Spalten für die Knoten der zweiten Menge. Unsere Matrix ist nun allerdings nicht mehr zwingend quadratisch, da beide Knotenmengen unterschiedlich groß sein können.

Beispiel für die Uniteile nach Wochentag.

Quellcode 11.6: Adjazenzmatrix für Uniteile pro Wochentag

Ort		StraNa	Reich.	Bibl.	Raab.
Tag	Idx	0	1	2	3
Mo.	0	X	X	-	-
Di.	1	X	-	-	-
Mi.	2	X	-	X	-
Do.	3	X	-	-	-
Fr.	4	-	X	-	-

Wollen wir die Aufenthaltszeiten am jeweiligen Tag abbilden, ordnen wir diese den Kanten als *Gewichte* zu, die Adjazenzmatrix nimmt nun `int`-Werte auf, wobei eine Aufenthaltsdauer 0 bedeutet, dass dieser Teil nicht besucht wird.

Quellcode 11.7: Adjazenzmatrix für Uniteile pro Wochentag mit Aufenthaltsdauer.

Ort		StraNa	Reich.	Bibl.	Raab.
Tag	Idx	0	1	2	3
Mo.	0	180	90	0	0
Di.	1	270	0	0	0
Mi.	2	90	0	240	0
Do.	3	180	0	0	0
Fr.	4	0	60	0	0

Wir kennen solche Anordnungen auch als Tabellenblätter. Die Zeilen beschreiben die einzelnen Objekte (Tage, Personen, Beobachtungen), die Spalten die verschiedenen beobachteten Werte (hier jeweils Zeiten, sonst auch Produktionsmengen, Messgrößen (Temperatur, Luftdruck, ...), Bewertungen), die oft auch unterschiedliche Datentypen haben können (bei verschiedenen Reisen z. B. Ziel, Dauer, Preis, Bewertung), was wir mit unserer Matrix allerdings nicht abbilden können – alle Daten müssen den gleichen Typ haben. Dann spricht man bei einer solchen Anordnung aus der Sicht der Statistik von einem *Data Frame*. In Python stellt die Bibliothek *Pandas* dafür eine angepasste Datenstruktur zur Verfügung.

Beispiel: Verwaltung der Uniorte zu bestimmten Tagen. In unsere Datendatei nehmen wir nun die Anzahl der Wochentage und der Orte auf und legen die Felder mit der korrekten Größe an. Aufbau der Datei:

- Anzahl Wochentage
- Wochentage, einer pro Zeile
- Anzahl Orte
- Orte mit Kennbuchstabe, Name, Wegzeit, durch Semikolon getrennt
- Matrix der Zeiten, jede Zeile enthält die Zeiten dieses Tages am jeweiligen Ort. Null bedeutet, dass der Ort nicht aufgesucht wird.

Beispieldatei:

Quellcode 11.8: studorte.dat: Daten der Uni-Orte

```

5
Montag
Dienstag
Mittwoch
Donnerstag
Freitag
4
A;Strasse der Nationen;15
C;Reichenhainer Straße;5
A;Bibliothek;20
B;Raabestrasse;40
180 90 0 0
270 0 0 0
90 0 240 0
180 0 0 0
0 60 0 0

```

Download: `studorte.dat`.

Programm zum Einlesen und Anzeigen der Daten:

Quellcode 11.9: studorte.c: Felder und Matrix für Verwaltung der Studienorte.

```

/* Adjazenzmatrix Wochentag-Veranstaltungsorte.
*/

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

enum {MAX_STR_LEN = 30, MAX_LINE_LEN = 50};

struct Place {
    char part;
    char name[MAX_STR_LEN];
    int timeDistance; // in Minuten
};

int readInt() {
    int res;
    scanf("%d", &res);
    return res;
}

/* Newline entfernen.
 * Es kann nur eines geben -> Überschreiben mit \0
 */
void removeNl(char s[]) {
    int i = 0;
    while (s[i] != '\n') i++;
    s[i] = '\0';
}

/* String rechts abschneiden.

```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```

*/
void leftStr(char dest[], char src[], int len) {
    for (int i = 0; src[i] != '\0' && i < len; i++) {
        dest[i] = src[i];
    }
    dest[len] = '\0';
}

void readDayNames(char dayNames[][MAX_STR_LEN+1], int numDays) {
    char line[MAX_LINE_LEN + 1];
    for (int i = 0; i < numDays; i++) {
        fgets(line, MAX_LINE_LEN, stdin);
        removeNl(line);
        strncpy(dayNames[i], line, MAX_STR_LEN);
    }
}

void showDayNames(char dayNames[][MAX_STR_LEN+1], int numDays) {
    for (int i = 0; i < numDays; i++) {
        printf("%2d: %s\n", i, dayNames[i]);
    }
}

void readPlaces(struct Place places[], int numPlaces) {
    char line[MAX_LINE_LEN + 1];
    for (int i = 0; i < numPlaces; i++) {
        fgets(line, MAX_LINE_LEN, stdin);
        char *token = strtok(line, ";");
        places[i].part = token[0]; // 1. Zeichen aus String der Länge 1
        token = strtok(NULL, ";");
        strncpy(places[i].name, token, MAX_STR_LEN);
        token = strtok(NULL, ";");
        places[i].timeDistance = atoi(token);
    }
}

void showPlace(struct Place place) {
    printf("%c: %s (Wegdauer: %d)\n", place.part, place.name,
        place.timeDistance);
}

void showPlaces(struct Place places[], int numPlaces) {
    for (int i = 0; i < numPlaces; i++) {
        showPlace(places[i]);
    }
}

void readTimeTable(int numDays, int numPlaces,
    int timeTable[numDays][numPlaces]) {
    for (int row = 0; row < numDays; row++) {
        for (int col = 0; col < numPlaces; col++) {
            timeTable[row][col] = readInt();
        }
    }
}
    
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```

void showTimeTable(int numDays, int numPlaces,
    char dayNames[][MAX_STR_LEN+1], struct Place places[],
    int timeTable[numDays][numPlaces]) {
    char abbr[5]; // Abgekürzte Tages- und Ortsnamen
    // Kopf
    printf("    |");
    for (int col = 0; col < numPlaces; col++) {
        leftStr(abbr, places[col].name, 4);
        printf(" %s", abbr);
    }
    printf("\n");
    printf("    |");
    for (int col = 0; col < numPlaces; col++) {
        printf(" %c ", places[col].part);
    }
    printf("\n");
    printf("----|");
    for (int col = 0; col < numPlaces; col++) {
        printf(" ----");
    }
    printf("\n");
    for (int row = 0; row < numDays; row++) {
        leftStr(abbr, dayNames[row], 2);
        printf("%s |", abbr);
        for (int col = 0; col < numPlaces; col++) {
            printf(" %4d", timeTable[row][col]);
        }
        printf("\n");
    }
}

int main() {
    int numDays = readInt();
    char dayNames[numDays][MAX_STR_LEN + 1];
    char line[MAX_LINE_LEN + 1];
    // newline entfernen
    fgets(line, MAX_LINE_LEN, stdin);
    readDayNames(dayNames, numDays);
    int numPlaces = readInt();
    struct Place places[numPlaces];
    int timeTable[numDays][numPlaces];
    // newline entfernen
    fgets(line, MAX_LINE_LEN, stdin);
    // Orte lesen
    readPlaces(places, numPlaces);
    // Adjazenzmatrix lesen
    readTimeTable(numDays, numPlaces, timeTable);
    printf("Liste der Wochentage:\n");
    showDayNames(dayNames, numDays);
    printf("\nListe der Uni-Orte:\n");
    showPlaces(places, numPlaces);
    printf("\nZeitplan:\n");
    showTimeTable(numDays, numPlaces, dayNames, places, timeTable);
}
    
```

Download: studorte.c.

Ergebnis für den Beispielatensatz:

Liste der Wochentage:

0: Montag
1: Dienstag
2: Mittwoch
3: Donnerstag
4: Freitag

Liste der Uni-Orte:

A: Strasse der Nationen (Wegdauer: 15)
C: Reichenhainer Straße (Wegdauer: 5)
A: Bibliothek (Wegdauer: 20)
B: Raabestrasse (Wegdauer: 40)

Zeitplan:

	Stras	Reic	Bibl	Raab
	A	C	A	B
Mo	180	90	0	0
Di	270	0	0	0
Mi	90	0	240	0
Do	180	0	0	0
Fr	0	60	0	0

Wir können nun verschiedene Auswertungen in diesem Graphen vornehmen. Als Beispiele betrachten wir:

- den Wochentag, an dem die meisten verschiedenen Orte aufgesucht werden,
- den Ort mit der geringsten Aufenthaltsdauer pro Woche,
- die mittlere Zeit pro Termin (= aufgesuchter Ort an einem bestimmten Wochentag).

Quellcode 11.10: Auswertungsfunktionen, zum Einfügen in studorte.c.

```

/* Auswertung der Zeitplanungstabelle.
 * Datenstruktur: int timeTable[numDays][numPlaces]
 */

/* Tag mit meisten verschiedenen Orten.
 */
int dayMaxPlaces(int numDays, int numPlaces,
    int timeTable[numDays][numPlaces]) {
    int dayMax = 0;
    int maxPlaces = 0;
    for (int day = 0; day < numDays; day++) {
        int cntPlaces = 0;
        for (int place = 0; place < numPlaces; place++) {
            if (timeTable[day][place] > 0) cntPlaces++;
        }
        if (cntPlaces > maxPlaces) {
            dayMax = day;
            maxPlaces = cntPlaces;
        }
    }
    return dayMax;
}

```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```

/* Ort, der am kürzesten aufgesucht wird.
 */
int minPlace(int numDays, int numPlaces,
             int timeTable[numDays][numPlaces]) {
    int placeMin = 0;
    int minTime = 0;
    // Initwert muss mind. so groß wie kleinster berechnete Wert sein!
    // -> berechne für 1. Ort
    for (int day = 0; day < numDays; day++) {
        // nicht besuchte Einträge haben Wert 0 und können
        // deshalb auch addiert werden.
        minTime += timeTable[day][placeMin];
    }
    for (int place = 1; place < numPlaces; place++) {
        int time = 0;
        for (int day = 0; day < numDays; day++) {
            time += timeTable[day][place];
        }
        if (time < minTime) {
            minTime = time;
            placeMin = place;
        }
    }
    return placeMin;
}

/* Mittlere Unidauer pro Termin.
 */
int meanTime(int numDays, int numPlaces,
             int timeTable[numDays][numPlaces]) {
    int time = 0; // alle Zeiten summieren
    int cntTimes = 0; // Anzahl aller Termine zählen
    for (int day = 0; day < numDays; day++) {
        for (int place = 0; place < numPlaces; place++) {
            if (timeTable[day][place] > 0) { // Nicht-Termine nicht zählen!
                time += timeTable[day][place];
                cntTimes++;
            }
        }
    }
    return ((double)time)/cntTimes;
}

```

Die Funktionen `dayMaxPlaces()` und `minPlace()` liefern uns als Rückgabe einen Tages- bzw. einen Ortsindex. Um den jeweiligen Tag oder Ort im Klartext zu erhalten, greifen wir auf die entsprechenden Felder `dayNames` bzw. `places` zu:

```

int i = dayMaxPlaces(numDays, numPlaces, timeTable);
printf("Tag mit höchster Ortszahl: %d %s\n", i, dayNames[i]);
i = minPlace(numDays, numPlaces, timeTable);
printf("Ort mit kürzester Wochenzeit: %d %c: %s\n",
       i, places[i].part, places[i].name);

```

11.4 Aufgaben

Aufgabe 11.1 *Bahnnetz*

Erweitern Sie das Programm der Bahnverbindungen. Für jede Verbindung soll die Entfernung und die Fahrzeit erfasst werden. Es soll eine Verbindungssuche möglich sein, wobei ein Umstieg erlaubt sein soll.

Aufgabe 11.2 *Bewertung Studienfächer, Hausaufgabe (8 Punkte)*

Es soll ein Programm erstellt werden, mit dem Sie Ihre Studienfächer nach verschiedenen Kriterien, wie Verständlichkeit der Vorlesung bzw. des Scripts, Qualität des Übungsmaterials, Qualität der Übungen, bewertet werden können. Für jedes Kriterium wird eine ganzzahlige Bewertung von 1 (schlecht) bis 5 (gut) vergeben, 0 bedeutet, dass keine Bewertung möglich ist.

Für jedes Studienfach werden erfasst:

- Name des Fachs (max. 30 Zeichen),
- Verantwortlicher (max. 30 Zeichen),
- Anzahl Semesterwochenstunden (1 SWS = 45 Minuten),
- Zeitfaktor Selbststudium

Die Daten werden extern erfasst und hintereinander in einer Datei mit folgendem Aufbau bereitgestellt:

- Liste der Fächer, ein Fach pro Zeile, Daten getrennt durch `^`, Abschluss mit der Zeile

```
###^###^0^0.0
```

- Liste der Kriterien (Text, max. 30 Zeichen), Abschluss mit der Zeile

```
###
```

- Tabelle der Bewertungen, pro Fach eine Zeile, Trennung der Werte durch Leerzeichen
- Liste von Kriterien, die weiter untersucht werden sollen (s. u.), abgeschlossen mit `###`.

Die Zahl der Fächer ist auf 30, die der Kriterien auf 20 begrenzt.

Beispieldatei: `ha1.dat` (mit 4 Veranstaltungen, 5 Kriterien, 4×5 Bewertungen und 2 Kriterien, die genauer untersucht werden sollen).

Ihr Programm soll eine beliebige Datei dieses Aufbaus per Dateiumlenkung einlesen.

Bearbeiten Sie folgende Teilaufgaben in Funktionen, rufen Sie diese aus der `main()`-Funktion auf und geben Sie jeweils das Ergebnis aus.

- Erzeugen einer sortierten Liste der Fächer, mit zusätzlicher Angabe des Selbststudienaufwands pro Woche in der Form **Stunden:Minuten**. Diese sortierte Liste soll in einer getrennten Funktion ausgegeben werden können.
- Erzeugen einer Tabelle, die zeilenweise die Fächer, sortiert nach Name, und spaltenweise die Bewertungen enthält, wobei die Bewertungen von links nach rechts alphabetisch sortiert sein sollen. Realisieren Sie Sortierung und Ausgabe in getrennten Funktionen!
- Ermitteln des Fachs mit der besten mittleren Bewertung, fehlende Bewertungen dürfen dabei nicht in Durchschnitt einbezogen werden.
- Ermitteln des Kriteriums, bei dem die meisten Bewertungen fehlen.
- Ermitteln des Kriteriums, das im Mittel am schlechtesten bewertet wurde.
- Ermitteln des Fachs, bei dem die meisten Bewertungen fehlen.

Für die Liste der Kriterien, die weiter untersucht werden sollen, sind folgende Informationen durch jeweils eine Funktion zu ermitteln und auszugeben:

- Der relative Anteil der Fächer, die bewertet wurden (also keine Bewertung 0 haben), als Wert zwischen 0 und 1.
- Die mittlere Bewertung für dieses Kriterium über alle bewerteten Fächer.
- Das Fach mit der schlechtesten Bewertung für dieses Kriterium.

Ausgaben sollten nicht in den Funktionen selbst erfolgen, sondern diese sollen einen geeigneten Wert an die `main()`-Funktion zurückliefern, und die `main()`-Funktion übernimmt die Ausgabe.

- *Aufbau von Python-Programmen*
- *Funktionen*
- *Variablen und Ausdrücke*
- *Bezeichner*
- *Verzweigungen*
- *Einfache Strings*
- *Ausgabe*
- *Eingabe*
- *Einfache Funktionen*
- *Schleifen*
- *Module und Standardbibliothek*
- *Kommandozeilenparameter und Typkonvertierung*
- *Aufgaben*

Auch für Python stellt der Rheinwerk-Verlag mit [Ernesti20] eine kostenlose Online-Version zur Verfügung. Eine systematische Einführung bietet [Kofler18]. Eine Umsetzung wichtiger Algorithmen und deren Prinzipien in Python zeigt [Kopec20] anhand von Beispielen.

12.1 Aufbau von Python-Programmen

Python-Programme bestehen aus einer Folge von Anweisungen, die zeilenweise angeordnet und gleichtief (oder überhaupt nicht) eingerückt sind. Sie werden *nicht* mit Semikolon abgeschlossen. Die Verwendung von Funktionen ist nicht zwingend nötig. Um jedoch den Code besser zu strukturieren, ist es sinnvoller, eine Menge von Funktionen zu definieren und den direkt ausgeführten Code auf den Aufruf einer Steuerfunktion zu beschränken. Dazu wird folgendes Muster empfohlen:

Quellcode 12.1: Empfohlener Python-Rahmen

```

"""
Rahmen für Python-Programme
"""

def run():
    """
    Steuerfunktion
    """
    pass # Platzhalter für Funktion

# Startcode
if __name__ == '__main__':
    run()

```

Der Vorteil dieses Musters ist neben der besseren Codestrukturierung und -lesbarkeit die Wiederverwendbarkeit. Wird diese Datei von einem anderen Python-Programm eingebunden, wird die `run()`-Funktion nicht automatisch ausgeführt, sondern nur, wenn sie explizit aufgerufen wird.

12.2 Funktionen

Funktionen werden wie folgt definiert:

```

def name(Parameterliste):
    [
        ...
        Dokumentation der Funktion
        ...
    ]
    Anweisungen
    [
        return [ausdruck]
    ]

```

Die Parameterliste enthält im einfachsten Fall einfach eine durch Komma getrennte Liste von Bezeichnern für jeden zu übergebenden Wert.

Alle Anweisungen eines Blocks *müssen* die gleiche Einrückung haben (sinnvoll: 2 oder 4 Leerzeichen, keine Tabulatoren verwenden). Mit der `return`-Anweisung wird der Funktionsruf beendet. Falls ein Ausdruck angegeben ist, wird dieser Wert zurückgegeben.

12.3 Variablen und Ausdrücke

Variablen verhalten sich in Python anders als in C, sie enthalten niemals das Objekt bzw. den Wert selbst, sondern nur einen Verweis darauf. Das Objekt selbst wird getrennt davon im Speicher abgelegt. Damit können Variablen zu verschiedenen Zeitpunkten auf verschiedene Objekte zeigen, die auch unterschiedliche Datentypen haben können. Die Benutzung erfolgt jedoch ähnlich wie in C, nur die Vereinbarung entfällt. Beispiele:

```

a = 42 # Anlegen eines int-Objekts mit Wert 42, a verweist darauf
a += 1 # Erzeugen eines int-Objekts mit Wert 43 und Ändern des Verweises von a

```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
b = a + 3
a = 'Hallo' # a verweist nun auf ein Objekt vom Typ string
```

Die arithmetischen Operatoren entsprechen denen von C, lediglich bei der Division gibt es Unterschiede: / liefert ein Fließpunktergebnis, auch wenn beide Operanden ganzzahlig sind (seit Python 3). Für die ganzzahlige Division muss der Operator // verwendet werden. Die Kurzoperatoren ++ und -- stehen nicht zur Verfügung, wohl aber die Operatoren += etc.

Python unterstützt Mehrfachzuweisungen der Form:

```
var1, var2 = ausdruck1, ausdruck2
```

Damit können z. B. die Inhalte zweier Variablen ohne Hilfsvariable einfach vertauscht werden:

```
a, b = b, a
```

Python unterstützt einen eigenen logischen Datentyp mit den beiden Literalen **False** und **True**. Für Tests müssen Ausdrücke vom logischen Datentyp verwendet werden, die Verwendung von anderen Typen ist nicht zulässig.

Vergleiche funktionieren weitestgehend identisch zu C, es werden die gleichen Operatoren verwendet. Die logischen Operatoren zum Verknüpfen mehrerer logischer Ausdrücke unterscheiden sich jedoch:

and Logisches Und

or Logisches Oder

not Logische Negation (unärer Operator)

12.4 Bezeichner

Für Bezeichner in Python gelten weitestgehend die gleichen Regeln wie für *Bezeichner* in C. Bezeichner, die mit einem Unterstrich beginnen, haben häufig eine Spezialbedeutung und sollten nicht für normale Variablen- und Funktionsnamen verwendet werden.

12.5 Verzweigungen

Binäre Verzweigungen werden mit der **if**-Anweisung realisiert:

```
if testausdruck :
    dannAnweisung
    ...
[
else:
    sonstAnweisung
    ...
]
```

Der Testausdruck muss nicht geklammert werden, aber die Zeile muss zur Einleitung des Blocks mit einem : abgeschlossen werden.

Beispiel: Bestimmung der Monatslänge unter Ausnutzung der Regelmäßigkeiten.

Quellcode 12.2: month.py: Monatslänge

```
def lenOfMonth(month):
    """
    Bestimmen der Monatslänge.

    :param `month`: Monatsnummer (1 = Januar)
    :returns: Monatslänge in Tagen
    """
    if month < 1 or month > 12:
        return -1
    if month == 2:
        return 28
    if (month < 7 and month % 2 == 0) or (month > 8 and month % 2) == 1:
        return 30
    else:
        return 31
```

Für Mehrfachverzweigungen mit if-else-if-Ketten steht das Schlüsselwort `elif` zur Verfügung, da aufgrund der Einrückungsregeln eine C-artige Schreibweise nicht möglich ist:

```
if testausdruck1 :
    dannAnweisung1
    ...
elif testausdruck2 :
    dannAnweisung2
    ...
...
[
else:
    sonstAnweisung
    ...
]
```

Eine mit `switch` vergleichbare Anweisung gibt es nicht.

12.6 Einfache Strings

Wir werden Strings in einem *Folgekapitel* genauer besprechen. Hier sollen wir nur kurz auf die notwendigen Informationen für Ein- und Ausgabe beschränken.

String-Literale werden wahlweise in Apostrophe oder Anführungszeichen eingeschlossen. Sie sind Objekte und können Variablen zugewiesen werden:

```
string1 = 'String in Apostrophen'
string2 = "String in Anführungszeichen"
```

Strings können beliebige Unicode-Zeichen enthalten, Umlaute dürfen also verwendet werden. Ein Test auf Gleichheit ist mit dem Operator `==` direkt möglich.

12.7 Ausgabe

Für einfache Ausgaben steht die Funktion `print()` zur Verfügung:

```
print(ausdruck [, ausdruck ...])
```

Die `print()`-Funktion erzeugt nach der Ausgabe des letzten Ausdrucks automatisch einen Zeilenumbruch. Wollen wir diesen abschalten, ergänzen wir den Parameter `end=''` (übergeben dem Parameter also einen leeren String).

Eine formatierte Ausgabe mit einem Musterstring ähnlich zu C ist mit folgendem Konstrukt möglich:

```
print('musterstring' % (parameterliste))
```

Ein einzelner Parameter kann auch ohne Klammern angegeben werden. Die Muster entsprechen weitestgehend denen von C.

Beispiele:

Quellcode 12.3: Klassische Stringformatierung

```
a = 42
pi = 3.14159265358979323
hallo = 'Schönen Guten Tag!'
wahr = True

print('Die Antwort ist %d' % a)
print('Die Antwort ist hexadezimal %x und oktal %o' % (a, a))
print('Die Kreiszahl  $\pi$  ist %f, gerundet %.2f' % (pi, pi))
print('Gruß: %s' % hallo)
print('Ist das wahr? %s' % wahr)
# Ausrichtung:
print('%-15s: %2d³ = %4d' % ('klein', 2, 2**3))
print('%-15s: %2d³ = %4d' % ('große Zahl', 10, 10**3))
```

Ergebnis:

```
Die Antwort ist 42
Die Antwort ist hexadezimal 2a und oktal 52
Die Kreiszahl  $\pi$  ist 3.141593, gerundet 3.14
Gruß: Schönen Guten Tag!
Ist das wahr? True
klein          :  2³ =    8
große Zahl     : 10³ = 1000
```

Bemerkung: Die Stringformatierung ist unabhängig von der Ausgabe. Die Formatierung erzeugt einfach ein neues Stringobjekt, das wir in einer Variable speichern oder einer Funktion übergeben können. Beispiel:

```
answerStr = '%s ist %d' % ('Die Antwort', 42)
```

Empfohlen wird jedoch ein neuerer, flexiblerer Mechanismus:

```
print('musterstring'.format(parameterliste))
```

Hier werden die Platzhalter durch das Klammernpaar `{}` dargestellt. Ohne weitere Angaben erfolgt die Standardformatierung entsprechend des Typs. Sollen Typumwandlungen erfolgen oder Feldlängen angegeben werden, werden diese innerhalb der Klammern nach einem Doppelpunkt angegeben.

```
{: [ausrichtung] [länge] formatzeichen }
```

Die Bedeutung der meisten Formatierungsangaben ist ähnlich wie in der klassischen Form, lediglich die Ausrichtung wird statt mit `-` mit `<` für linksbündige und `>` für rechtsbündige Ausrichtung (Vorgabe) angegeben.

Beispiele:

```
a = 42
pi = 3.14159265358979323
hallo = 'Schönen Guten Tag!'
wahr = True

print('Die Antwort ist {}'.format(a))
print('Die Antwort ist hexadezimal {:x} und binär {:b}'.format(a, a))
print('Die Kreiszahl  $\pi$  ist {}, gerundet {:.2f}'.format(pi, pi))
print('Gruß: {}'.format(hallo))
print('Ist das wahr? {}'.format(wahr))
# Ausrichtung:
print('{:<15s}: {:>2d}^3 = {:>4d}'.format('klein', 2, 2**3))
print('{:<15s}: {:>2d}^3 = {:>4d}'.format('große Zahl', 10, 10**3))
```

Ergebnis (identisch mit der alten Form außer der neuen Möglichkeit der Binärausgabe):

```
Die Antwort ist 42
Die Antwort ist hexadezimal 2a und binär 101010
Die Kreiszahl  $\pi$  ist 3.141592653589793, gerundet 3.14
Gruß: Schönen Guten Tag!
Ist das wahr? True
klein          :  23 =    8
große Zahl     : 103 = 1000
```

Bemerkung: In der Formatangabe darf vor dem Doppelpunkt ein Identifier für das entsprechende Feld stehen, auf den in den Formatparametern Bezug genommen werden kann. Das ermöglicht unterschiedliche Anordnungen der Bestandteile, wenn das Formatmuster aus z. B. einer sprachabhängigen Variablen stammt. Darauf soll hier nicht weiter eingegangen werden.

12.8 Eingabe

Zum Lesen einer Zeile von der Konsole stellt uns Python die Funktion `input()` zur Verfügung, die eine komplette Zeile liest und uns diese als String zur Verfügung stellt.

Eine vergleichbare Funktionalität wie die Funktionen `readInt()` und `readDouble()` können wir mit folgendem Python-Modul erreichen:

```
"""
Lesen des nächsten Wertes von der Konsole.
Nutzung:
* Datei ins Projektverzeichnis kopieren
* Am Anfang des Codes einfügen:
  from read import Read
* Nutzung::
  x = Read.nextInt()
  y = Read.nextFloat()
  z = Read.nextBool()
  s = Read.nextStr() # nur ohne Leerzeichen!
"""

class Read:
    line = list()

    def nextVal() -> str:
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```

while Read.line is not None and len(Read.line) == 0:
    try:
        txt = input().strip()
        Read.line = txt.split()
    except EOFError:
        Read.line = None
if Read.line is not None:
    val = Read.line[0]
    Read.line = Read.line[1:]
else:
    val = None
return val

def nextInt() -> int:
    s = Read.nextVal()
    if s is not None:
        return int(s)
    else:
        return None

def nextFloat() -> float:
    s = Read.nextVal()
    if s is not None:
        return float(s)
    else:
        return None

def nextBool() -> bool:
    s = Read.nextVal()
    return s in ('True', 'true', 't', 'T', '1')

def nextStr() -> str:
    """
    Alias für nextVal()
    """
    return Read.nextVal()
    
```

Download: read.py.

Beispiel: Einlesen einer Folge ganzer Zahlen bis -1 und berechnen der Summe (ohne die -1):

```

from read import Read

def sumSeq() -> int:
    """
    Einlesen einer Zahlenfolge bis -1 (excl.) und Summe dieser Folge
    """
    partSum = 0
    val = Read.nextInt()
    while val != -1:
        partSum += val
        val = Read.nextInt()
    return partSum

def run():
    """
    
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```

Steuerprogramm.
Summenberechnung und Ausgabe.
"""

mySum = sumSeq()
print("Summe: {}".format(mySum))

if __name__ == '__main__':
    run()
    
```

12.9 Einfache Funktionen

In Python werden Funktionen mit dem Schlüsselwort **def** definiert:

```

def funktionsName( [parameterliste] ):
    Block
    ...
    [
        return [ausdruck]
    ]
    
```

Alle Zeilen des Blocks müssen gleichtief eingerückt sein. Die Parameterliste enthält einfach die Namen der lokalen Parameter, eine Typangabe entfällt. Ebenso wird kein Rückgabetyt spezifiziert.

Bemerkung: Seit Python 3.5 werden sogenannte Type hints ([PEP 484](#)) unterstützt. Durch den Python-Interpreter werden diese allerdings nicht ausgewertet, dazu werden Zusatzwerkzeuge benötigt. Auch ohne Prüfung ist ihr Einsatz zur besseren Dokumentation des Quelltextes sinnvoll.

Beispiel: Berechnungen für gleichmäßig beschleunigte Bewegungen. Zur Demonstration wird die Wurzelfunktion ebenfalls selbst implementiert (sie steht im Modul `math` unter dem Namen `sqrt()` zur Verfügung).

Quellcode 12.4: `accel.py`: Funktionen gleichmäßig beschleunigte Bewegung

```

"""
Funktionen beschleunigte Bewegung
"""

def sqrt(a):
    """
    Wurzel aus x.
    In Modul math vorhanden, hier zur Demonstration mit Heron-Verfahren.
    """

    eps = 1e-6 # Genauigkeit
    xOld = a
    xNew = a + 2 * eps
    while abs(xOld - xNew) > eps: # abs() ist eine eingebaute Funktion
        xOld = xNew
        xNew = (xOld + a/xOld)/2
    return xNew

def t2v(t, g=9.81):
    """
    
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```

Geschwindigkeit (m/s) aus Zeit (s):
:param t: Zeit (s)
:param g: Beschleunigung (m/s^2)
:return: Geschwindigkeit (m/s)
"""
return g * t;

def t2s(t, g=9.81):
    """
    Strecke (m) aus Zeit (s):
    :param t: Zeit (s)
    :param g: Beschleunigung (m/s^2)
    :return: Strecke (m)
    """
    return 0.5* g * t * t;

def s2t(s, g=9.81):
    """
    Zeit aus Strecke
    :param s: Strecke (m)
    :param g: Beschleunigung (m/s^2)
    :return: Zeit (s)
    """
    return sqrt(2 * s/g)

def run():
    """
    Smoke test der Funktionen
    """
    t = 10
    v = t2v(t)
    s = t2s(t)
    print('Bei {} s: v = {:.2f} m/s, s = {:.2f} m'.format(t, v, s))
    # Probe
    tt = s2t(s)
    print('Probe: t = {:.2f} s'.format(tt))
    gMond = 1.62
    print('Für den Mond (g = {})'.format(gMond))
    v = t2v(t, gMond)
    s = t2s(t, gMond)
    print('Bei {} s: v = {:.2f} m/s, s = {:.2f} m'.format(t, v, s))
    tt = s2t(s, gMond)
    print('Probe: t = {:.2f} s'.format(tt))

if __name__ == '__main__':
    run()
    
```

Ausgabe:

```

Bei 10 s: v = 98.10 m/s, s = 490.50 m
Probe: t = 10.00 s
Für den Mond (g = 1.62)
Bei 10 s: v = 16.20 m/s, s = 81.00 m
Probe: t = 10.00 s
    
```

 Download: [accel.py](#).

Parameter werden an Python-Funktionen prinzipiell als *Referenz* übergeben. Da jedoch alle einfachen Objekte und auch Strings unveränderliche Objekte sind, gleicht das Verhalten dem Aufruf mittels Wert.

Beispiel (*a* sei eine ganze Zahl):

```
def f(a):  
    a = a + 1  
    return a
```

Rufen wir diese Funktion mit einer Variablen *b* = 1 auf, enthält *b* eine Referenz auf das Objekt 1. Innerhalb der Funktion zeigt *a* zunächst auf das gleiche Objekt. Der Ausdruck *a* = *a* + 1 berechnet die Summe aus dem Objekt, auf das *a* zeigt, und 1 und erzeugt daraus ein neues Objekt. Die Variable *a* zeigt nach der Zuweisung auf dieses Objekt, das ursprüngliche Objekt bleibt unverändert, *b* zeigt also weiter auf 1.

Übergeben wir dagegen Listen, Dictionaries oder eigene komplexe Objekte an eine Funktion, kann die Funktion meist den Inhalt der Originalobjekte verändern, wie wir das bei Feldern in C gesehen haben.

Python unterstützt neben Positionsparametern wie in C auch mit Werten vorbelegte Parameter der Form

`name = wert`

In der Funktionsdefinition müssen diese Parameter rechts stehen, nach einem vorbelegten Parameter darf kein unbelegter Parameter stehen. Beim Aufruf einer solchen Funktion werden entweder alle Parameter angegeben, dann benutzt die Funktion die übergebenen Werte. Werden beim Aufruf dagegen Parameter weggelassen, dann werden die Funktionsparameter von rechts nach links mit den Vorgabewerten belegt und diese verwendet.

Beispiel: Fallhöhe in einer gewissen Zeit. Diese ist von der Beschleunigung abhängig. Wir verwenden hier den Vorgabewert 9.81 (m/s^2) für die Erdbeschleunigung. Wollen wir die Fallhöhe (oder besser die zurückgelegten Strecke) auf einer schiefen Ebene oder auf einem anderen Planeten bestimmen, übergeben wir den abweichenden Wert der Beschleunigung.

```
def distFreeFall(t, a = 9.81):  
    """  
    Strecke, die in Zeit t zurückgelegt wird.  
    :param t: Zeit in Sekunden  
    :param a: Beschleunigung (m/s2), Vorgabe: 9.81  
    :return: Strecke  
    """  
    return a * t * t / 2
```

Aufruf:

```
distFreeFall(3)      # = 44.145  
distFreeFall(3, 2)  # = 9.0
```

Die Parameternamen können auch beim Funktionsaufruf spezifiziert werden, damit kann die Reihenfolge der Parameter beim Aufruf modifiziert werden:

```
distFreeFall(t = 4)      # = 78.48  
distFreeFall(a = 5, t = 2) # = 10.0
```

Wir haben die Nutzung der Vorgabewerte bereits bei der `print()`-Funktion praktisch gesehen (Parameter `end`, der normalerweise eine Zeilenschaltung bewirkt).

Die Verwendung benannter Funktionsparameter erlaubt es auch, beim Aufruf vordefinierte Parameter zu »überspringen«, d. h. nur diese Parameter mit eigenen Werten zu belegen, die wir tatsächlich verändern wollen.

Eine Python-Funktion kann mehrere Werte zurückgeben:

```
return ausdruck1 [, ausdruck2 ...]
```

Der Aufruf der Funktion mit Verarbeitung der Funktionswerte kann dann wie folgt geschehen:

```
variable1 [, variable2 ...] = funktionsName(parameterListe)
```

Beispiel: Lösungen einer quadratischen Gleichung.

Quellcode 12.5: solveqeq.py: Lösen einer quadratischen Gleichung
(mit Probe)

```
def absDiff(x, y):
    diff = x - y
    if diff < 0:
        diff = -diff
    return diff

# Wurzelfunktion im Modul math verfügbar
def sqrt(a, xOld = None, EPS=1e-6):
    """
    Rekursive Wurzelberechnung.
    :param a: Radikant
    :param xOld: beim ersten Aufruf None, sonst voriger Näherungswert
    :param EPS: Genauigkeit
    """
    if xOld is None: # erster Aufruf: setze xOld = a
        return sqrt(a, a, EPS)
    x = (xOld + a/xOld)/2 # neuer Näherungswert
    if absDiff(x, xOld) < EPS: # Genauigkeit erreicht?
        return x
    else:
        return sqrt(a, x, EPS)

def solveQeq(a, b, c, EPS=1e-6):
    """
    Löse  $ax^2 + bx + c = 0$ 
    :return: x1, x2 oder None, None, wenn nicht lösbar
    """
    d = b * b - 4 * a * c
    if d < 0:
        return None, None
    rootD = sqrt(d, EPS) # Parameter xOld ausgelassen, deshalb benannter Parameter
    return (-b - rootD)/(2 * a), (-b + rootD)/(2 * a)

def poly2(x, a, b, c):
    return ((a * x) + b) * x + c
```

Download: solveqeq.py. Aufruf der Funktion:

```
a, b, c = 2, -5, 3 # Testdaten Parameter
x1, x2 = solveQeq(a, b, c)
if x1 is not None:
    # Verarbeite x1, x2, z. B. Probe:
    print(poly2(x, a, b, c))
# Höhere Genauigkeit
x1, x2 = solveQeq(a, b, c, EPS=1e-12)
# ...
```

Das Literal **None** steht für ein Spezialobjekt, das den Wert »Nichts« repräsentiert. Der Test auf diesen Wert ist mit den logischen Operatoren **is None** bzw. **is not None** möglich (ein Vergleich **== None** bzw. **!= None** funktioniert ebenso, wird aber nicht empfohlen).

12.10 Schleifen

Python stellt eine abweisende Schleife zur Verfügung, die sich weitestgehend wie ihr Pedant in C verhält:

```
while schleifenTest:
    Block
```

Die Schleife wird solange abgearbeitet, bis der Schleifentest fehlschlägt.

Bemerkung: Die Schleife kann noch um einen **else**-Zweig erweitert werden, der nach Verlassen der Schleife abgearbeitet wird, wenn die Schleife nicht vorzeitig durch ein **break** verlassen wird.

Beispiel: Intervallsumme von a bis b (vergl. *Intervallsumme*).

```
def sum(a, b):
    partSum = 0
    while a <= b:
        partSum += a
        a += 1 # a++ in Python nicht möglich
    return partSum
```

Die **for**-Schleife hat in Python jedoch eine andere Semantik: Sie *iteriert* über eine Menge von Werten, z. B. alle Werte einer Liste bzw. Feldes.

```
for wertVariable in wertMenge:
    Block
```

Um eine Zählschleife zu erzeugen, benutzen wir die Funktion **range()** zum Erzeugen einer Zahlenfolge:

```
bereich = range([ start, ] ende [ , schrittweite ] )
```

Geben wir nur den Endwert an, wird eine Folge von Zahlen von 0 bis **ende-1** erzeugt. Mit Angabe eines Startwertes beginnt die Folge erst bei dieser Zahl. Mit der Angabe einer Schrittweite können Werte übersprungen oder auch bei einem negativen Wert in umgekehrter Reihenfolge erzeugt werden. Wichtig ist, dass der Startwert immer, der Endwert dagegen nie zur Folge gehört.

Beispiele:

```
range(3)          # → (0, 1, 2)
range(1, 3)       # → (1, 2)
range(2, 10, 2)   # → (2, 4, 6, 8)
range(5, 0, -1)   # → (5, 4, 3, 2, 1)
```

Bemerkung: Um die generierten Bereiche zur Kontrolle anzuzeigen, müssen wir sie in eine Liste umwandeln:

```
print(list(range(2, 10, 2)))
```

Eine Zählschleife erzielen wir wie folgt:

```
for zaehlVariable in range(grenzen):
    # Verarbeite zaehlVariable
    Block
```

Unsere Intervallsumme können wir mit Hilfe einer **for**-Schleife wie folgt berechnen:

```
def sum(a, b):
    partSum = 0
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
for value in range(a, b + 1):
    partSum += value
return partSum
```

Die Schleifensteuerung mittels **break** und **continue** ist sowohl bei **while**- als auch bei **for**-Schleifen wie in C möglich. Zusätzlich bietet Python die Möglichkeit mit Hilfe des **else**-Zweiges der Schleifen nach Schleifenende festzustellen, ob die Schleife mittels **break** abgebrochen wurde oder komplett bis zum fehlgeschlagenen Schleifentest durchgelaufen ist.

12.11 Module und Standardbibliothek

Wollen wir Funktionen in getrennten Dateien definieren, um diese in verschiedenen Python-Programmen nutzen zu können, dann müssen wir diese importieren:

```
from basisName import objekt1 [, objekt2 :ms:...]
```

Der **basisName** ist der Dateiname mit der oder den benötigten Definitionen ohne die Dateierweiterung (üblicherweise **.py**). Die Objekte sind üblicherweise Funktions- oder Klassennamen, können aber auch andere Objekte wie Variablen und Konstanten sein. Wir können anschließend die aufgelisteten Objekte in unserem Programm verwenden.

Beispiel: Verwendung der Funktion **solveQeq()** aus *solveqeq.py: Lösen einer quadratischen Gleichung (mit Probe)*.

```
from solveqeq import solveQeq

x1, x2 = solveQeq(1, -2, 1)
```

Um alle in einer Datei definierten Objekte verfügbar zu machen, können wir diese auch komplett importieren:

```
import basisName
```

Die in dieser Datei definierten Objekte müssen bei ihrer Verwendung nun jedoch durch Voranstellen des Basisnamens als Präfix qualifiziert werden:

```
basisName.objektName
```

Beispiel für den kompletten Import aller Funktionen für quadratische Gleichungen:

```
import solveqeq

x1, x2 = solveqeq.solveQeq(1, -2, 1)
# Differenz der Lösungen
diff = solveqeq.absDiff(x1, x2)
```

Dateien mit nützlichen Funktionen, die von anderen Programmen verwendet werden können, nennt man *Module*. Diese können hierarchisch organisiert werden (worauf wir hier nicht eingehen). Sie werden entweder in speziellen Modulverzeichnissen oder im Arbeitsverzeichnis gesucht.

Die Python-Installation stellt eine umfangreiche Standardbibliothek in Form von Modulen zur Verfügung. Diese sind unter

<https://docs.python.org/3/library/>

dokumentiert. Die mathematischen Funktionen finden sich z. B. im Modul **math** ([Dokumentation](#)).

Beispiel:

```
import math

# Erdradius
r = 6366.2
# Erdumfang
u = r * math.pi
# Abstand Chemnitz (50.8-ter Breitengrad) vom Äquator
dist = r * math.radians(50.8) # Umrechnung Grad → Radiant
# Senkrechter Abstand zur Äquatorebene
distVert = r * math.sin(math.radians(50.8))
# Diagonale eines Rechtecks mit Seitenlängen 10 und 12
diag = math.sqrt(10**2 + 12**2)
# oder
diag = math.hypot(10, 12)
```

Bemerkung: Die Python-Standardbibliothek ist deutlich umfangreicher als die C-Standardbibliothek.

12.12 Kommandozeilenparameter und Typkonvertierung

Um in Python Kommandozeilenparameter zu lesen, müssen wir das Modul `sys` importieren, in dem uns diese als Liste `argv` bereitgestellt wird. Das erste Listenelement (Index 0) ist der Name des Python-Scripts, die folgenden sind die übergebenen Parameter. Ein Äquivalent zu `argc` wird nicht benötigt, da wir mit der Funktion `len()` die Anzahl der Parameter einfach abfragen können.

Bemerkung: Der beim Aufruf vor dem Scriptnamen stehende Python-Aufruf taucht nicht in `sys.argv` auf.

Beispiel: Anzeige aller übergebenen Parameter.

Quellcode 12.6: `commandline.py`: Anzeige der Kommandozeilenparameter

```
"""
Anzeige der übergebenen Kommandozeilenparameter
"""
import sys

def run():
    print('Anzahl Parameter (incl. Programmname): {}'.format(len(sys.argv)))
    print('\nVariante 1:\n')
    for i in range(len(sys.argv)):
        print("{}: {}".format(i, sys.argv[i]))
    # Alternativ mit Iterator
    print("\nAlternativ:\n")
    for param in sys.argv:
        print(param)

if __name__ == '__main__':
    run()
```

Download `commandline.py`. Ergebnis für den Aufruf `python commandline.py` Antwort 42 True:

```
Anzahl Parameter (incl. Programmname): 4
```

Variante 1:

```
0: commandline.py
1: antwort
2: 42
3: True
```

Alternativ:

```
commandline.py
antwort
42
True
```

Die Parameter werden prinzipiell als Strings übergeben. Wollen wir diese einfach wie Zahlen benutzen, erhalten wir einen Typfehler.

Beispiel:

Quellcode 12.7: add_err.py: Versuch, alle Kommandozeilenparameter zu addieren

```
"""
Addition aller Kommandozeilenparameter, die als
Zahlen interpretiert werden sollen.
"""

import sys

def run():
    sum = 0
    for param in sys.argv[1:]:
        sum += param
    print(sum)

if __name__ == '__main__':
    run()
```

Ergebnis für den Aufruf `python add_err 1 2 3`:

```
Traceback (most recent call last):
  File "add_err.py", line 15, in <module>
    run()
  File "add_err.py", line 11, in run
    sum += param
TypeError: unsupported operand type(s) for +=: 'int' and 'str'
```

Wir müssen die Strings explizit in den benötigten Typ umwandeln:

`int(string)` Umwandlung des Strings in eine ganze Zahl.

`float(string)` Umwandlung des Strings in eine Gleitpunktzahl.

Für den Typ `bool` funktioniert dieser Mechanismus allerdings nicht, wir erhalten für jeden beliebigen nichtleeren String den Wert `True`. Hier können wir folgende Konstruktion verwenden:

```
def str2bool(string):
    """
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
Liefert für die Strings True, true, T, t und 1 den Wert True, sonst False.
"""
return string in ('True', 'true', 'T', 't', '1')
```

Beispiel: Korrekte Addition.

Quellcode 12.8: add.py: Addition aller Kommandozeilenparameter

```
"""
Addition aller Kommandozeilenparameter, die als
Zahlen interpretiert werden sollen.
"""

import sys

def run():
    sum = 0
    for param in sys.argv[1:]:
        sum += param
    print(sum)

if __name__ == '__main__':
    run()
```

Beispiel: Berechnung der Potenzfunktion. Die Basis darf eine Gleitpunktzahl sein, der Exponent ist eine ganze Zahl.

Quellcode 12.9: power.py: Potenzfunktion

```
import sys

def power(base, exp):
    """
    Potenzfunktion: base hoch exp
    :return: Funktionswert
    """
    if exp == 0:
        return 1
    if exp < 0: # Spezialfall exp < 0: a-b = 1/ab
        return 1.0/power(base, -exp)
    return base * power(base, exp - 1)

def usage():
    # Zugriff auf Programmnamen: sys.argv[0]
    print('Usage {} base exp'.format(sys.argv[0]))
    print('  base: Gleitpunktzahl')
    print('  exp: Ganzzahl')

def run():
    if len(sys.argv) != 3:
        usage()
        sys.exit(1) # Programm mit Exit-Code 1 abbrechen
    # Kommandozeilenparameter lesen und konvertieren
    base = float(sys.argv[1])
    exp = int(sys.argv[2])
    res = power(base, exp)
    print('{}^{} = {}'.format(base, exp, res))
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
if __name__ == '__main__':  
    run()
```

Download: `power.py`.

Bemerkung: Sehr komfortable und flexible Möglichkeiten zum Verarbeiten von Kommandozeilenparametern stellt das Modul `argparse` zur Verfügung.

12.13 Aufgaben

Implementieren Sie die *Schleifen-Aufgabe* nun in Python.

- *Überblick*
- *Erzeugen, Elementzugriff*
- *Übergabe an Funktionen*
- *Mehrdimensionale Listen*
- *Verknüpfen von Listen*
- *Aufgaben*

13.1 Überblick

Python kennt zunächst keinen Feldtyp im C-Sinne, sondern verwendet eine erweiterte Datenstruktur **list**. Im Gegensatz zu Feldern können Listen dynamisch wachsen und wieder verkürzt werden.

13.2 Erzeugen, Elementzugriff

Eine Liste mit vorgegebener Länge kann wie folgt erzeugt werden:

```
listVar = [0]*laenge
```

Wir erhalten eine Liste der angegebenen Länge, deren Elemente alle den Wert 0 haben. Wir können auch alle Werte einzeln aufzählen:

```
listVar = [wert0, ..., wert_n]
```

Alternativ und recht flexibel können Listen mit dem **for**-Konstrukt erzeugt werden:

```
[ausdruck for variable in menge]
```

Die Variable durchläuft dabei alle Elemente der Menge. Für jedes dieser Elemente wird ein Listenelement mit dem Wert **ausdruck** erzeugt und an die Liste angehängt. Der Ausdruck enthält deshalb oft die Variable zur Berechnung des jeweiligen Listenwertes.

Beispiele:

```
ones = [1]*23                # Liste mit 23-mal der Zahl 1
ones = [1 for i in range(23)] # alternative for-Syntax
fib = [1, 1, 2, 3, 5, 8]     # Liste der ersten 6 Fibonacci-Zahlen
squares = [x*x for x in range(1, 11)] # Quadratzahlen von 12 bis 102
```

Ein Zugriff erfolgt mit der gleichen Syntax wie in C:

```
element = listVar[index]
listVar[index] = neuerWert
```

Der Index darf auch negativ sein, dann werden die Elemente von hinten abgezählt. Der Listenindex $-i$ bedeutet das gleiche wie `len(listVar)-i`. Der Zugriff

```
squares[-1]
```

liefert also das letzte Listenelement, im Beispiel also `squares[9]` mit dem Wert 100.

Die Länge einer Liste kann mit der Funktion `len()` abgefragt werden:

```
laenge = len(liste)
```

Um alle Elemente einer Liste zu verarbeiten, können wir ähnlich wie in C eine Schleife über die Indices der Listenelemente schreiben. Um alle Quadratzahlen des obigen Beispiels auszugeben, verwenden wir:

```
for i in range(len(squares)):
    print(squares[i])
```

Eleganter ist jedoch, die Schleife die Listenelemente selbst auswählen zu lassen:

```
for element in liste:
    # Verarbeite element
    ...
```

Beispiel für die Liste der Quadratzahlen:

```
for square in squares:
    print(square)
```

Benötigen wir zusätzlich den Index der Elemente, iterieren wir nicht über die Liste selbst, sondern kapseln sie in die Funktion `enumerate()`:

```
for index, element in enumerate(liste):
    # Verarbeite index und element
    ...
```

Beispiel: Ausgabe der Fibonacci-Zahlen mit ihrem Index:

```
for i, number in enumerate(fib):
    print(i, number)
```

Das Verlängern einer Liste ist mit der Funktion (eigentlich Methode) `append()` möglich:

```
listVar.append(element)
```

Bemerkung: Eine Methode (siehe *Python-Klassen*) verhält sich wie eine Funktion, ist jedoch an ein konkretes Objekt, hier eine Liste, gebunden und wird an diese, durch einen Punkt getrennt, angehängt.

Da mit der Methode `append()` die Länge einer Liste vergrößert werden kann, ist das Anlegen einer leeren Liste (mit 0 Einträgen) ebenfalls eine nützliche Operation. Dies kann entweder mittels

```
listVar = []
```

oder

```
listVar = list()
```

erfolgen.

Um ein Element an einer beliebigen Stelle in die Liste einzufügen, verwenden wir die Methode `insert()`:

```
listVar.insert(position, element)
```

Zum Löschen eines Elements aus einer Liste dient die Methode `pop()`:

```
listVar.pop(position)
```

Beispiele:

```
primes = [2, 3, 5, 7, 11]
# nächste Primzahl anhängen
primes.append(13) # → [2, 3, 5, 7, 11, 13]
# in Liste ungerader Zahlen umbauen: 2 weg, 9 hinzu
primes.insert(4, 9) # → [2, 3, 5, 7, 9, 11, 13]
primes.pop(0) # → [3, 5, 7, 9, 11, 13]
# Primzahlliste wieder herstellen
primes.pop(3) # → [3, 5, 7, 11, 13]
primes.insert(0, 2) # → [2, 3, 5, 7, 11, 13]
```

Wir können aus einer Liste einfach Teillisten auswählen:

```
listVar[startIndex:endIndex]
```

Es entsteht eine Teilliste (Kopie), die alle Elemente einschließlich des Startindex und ausschließlich des Endindex enthält. Werden einer oder beide Indices weggelassen, ist der Startindex 0 und der Endindex die Länge der Liste.

Beispiele:

```
fib = [1, 1, 2, 3, 5, 8]
f1 = fib[2:4] # → [2, 3]
f2 = fib[:4] # → [1, 1, 2, 3]
f3 = fib[4:] # → [5, 8]
f4 = fib[:-2] # → [1, 1, 2, 3]
f5 = fib[:] # → [1, 1, 2, 3, 5, 8] (Kopie)
```

13.3 Übergabe an Funktionen

Listen können problemlos an Funktionen übergeben werden, ebenso können Funktionen Listen zurückgeben.

Die Funktion erhält eine *Referenz* auf die Liste. Da Listen veränderbar sind, kann die Funktion die Originalliste verändern. Ist dies unerwünscht, müssen wir explizit eine Kopie der Liste erzeugen:

```
kopie = liste[:]
```

Beispiel: Differenz zweier Vektoren `v1` und `v2`. Wir erzeugen zunächst eine Kopie von `v1`, subtrahieren elementweise `v2` und geben den kopierten und veränderten Vektor zurück.

```
def vectorDiff(v1, v2):
    """
    Differenz zweier Vektoren
    :return: Differenzvektor
    """
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```

if len(v1) != len(v2):
    return None
# Kopiere v1
res = v1[:]
for i in range(len(res)):
    res[i] -= v2[i]
return res

def test():
    """
    Test der prinzipiellen Funktionalität
    """
    a = [2, 3, 2, 8]
    b = [1, 4, 8, 2]
    c = [1, 4, 8, 2, 5]

    print(vectorDiff(a, b))
    print(vectorDiff(a, c))
    
```

13.4 Mehrdimensionale Listen

Wir können mehrdimensionale Felder als Liste anlegen, die wiederum Listen als Elemente enthält, z. B.

```

feld = [
    [0, 1, 2, 3],
    [4, 5, 6, 7],
    [8, 9, 10, 11]]
    
```

Wir erhalten ein Feld mit 3 Zeilen und 4 Spalten. Der Zugriff auf Feldelemente erfolgt wie gewohnt mittels

```
feld[zeilenIndex][spaltenIndex]
```

Im Beispiel liefert der Zugriff `feld[1][2]` den Wert 6.

Wollen wir zweidimensionales Feld erzeugen, dass mit einem festen Wert initialisiert ist, müssen wir jedoch etwas aufpassen. Das Konstrukt

```
feld = [[0]*3]*2
```

erzeugt zwar ein Feld mit 2 Zeilen und 3 Spalten, wir erhalten

```
[[0, 0, 0], [0, 0, 0]]
```

als Ausgabe. Verändern wir jedoch ein einzelnes Element, z. B.

```
feld[1][2] = 42
```

und zeigen das Feld erneut an, erhalten wir

```
[[0, 0, 42], [0, 0, 42]]
```

Der Grund ist, dass die »äußere« Liste zwei Referenzen auf die gleiche innere Liste, also das gleiche Objekt enthält.

Wir müssen deshalb etwas mehr Aufwand investieren, damit wirklich jede Zeile eine eigene Zeile ist:

```
# Zweidimensionales Feld mit 2 Zeilen und 3 Spalten

feld = [[0]*3 for _ in range(2)]

# oder

feld = [[0, 0, 0] for _ in range(2)]

# oder mit gleichem Konstrukt für innere Liste

feld = [[0 for _ in range(3)] for _ in range(2)]
```

Die Dimensionen eines zweidimensionalen Feldes können wir wie folgt ermitteln:

Zeilenzahl: `len(feld)`, da `feld` ja eine Liste aller Zeilen enthält.

Spaltenzahl: `len(feld)[0]`. Wir holen uns die erste Zeile des Feldes und bestimmen ihre Länge. Voraussetzung ist natürlich, dass das Feld mindestens eine Zeile enthält.

Beispiel: Multiplikationstabelle. Die Größe kann als Kommandozeilenparameter übergeben werden.

Quellcode 13.1: multtable.py: Multiplikationstabelle

```
import sys

def multTable(n):
    """
    Multiplikationstabelle von 0*0 bis n*n
    :return: Das von der Funktion erzeugte Feld
    """

    table = [[0 for _ in range(n+1)] for _ in range(n+1)]
    for i in range(n+1):
        # nur oberes Dreieck erzeugen und spiegeln
        for k in range(i, n+1):
            table[i][k] = table[k][i] = i * k
    return table

def showTable(table):
    """
    Tabelle mit Beschriftung anzeigen
    """

    n = len(table) # Zeilenzahl, Spaltenzahl ist identisch
    # Spaltenköpfe
    print('    |', end='')
    for col in range(n):
        print('{:4d}'.format(col), end='')
    print()
    print('----|', end='')
    for col in range(n):
        print(' ---', end='')
    print()
    # Tabelle
    for row in range(n):
        print('----|', end='')
        for col in range(n):
            print('{:4d}'.format(table[row][col]), end='')
        print()
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
def run():
    n = 10
    if len(sys.argv) == 2:
        n = int(sys.argv[1])
    table = multTable(n)
    showTable(table)

if __name__ == '__main__':
    run()
```

Benötigen wir eine komplette Zeile eines zweidimensionalen Feldes, können wir diese als Liste erhalten:

```
feld[zeilenIndex]
```

Beispiel: Matrixmultiplikation. Wir lesen beide Matrizen mit ganzzahligen Werten aus einer Datei mit folgendem Aufbau ein:

```
zeilenA spaltenA
WerteVonAzeilenweise

zeilenB spaltenB
WerteVonBzeilenweise
```

Beispieldatensatz: matmul1.dat (A: 3×4-Matrix, B: 4×2-Matrix).

```
3 4
1 2 3 4
5 6 7 8
9 10 11 12

4 2
5 7
1 3
2 8
9 1
```

Wir zeigen diese Matrizen zunächst an. Hier benutzen wir statt des Indexzugriffs `feld[zeile][spalte]` mit zwei Schleifen über die Indices folgende Alternative:

```
for zeile in feld:
    for wert in zeile:
        print(wert)
```

Für die eigentliche Matrixmultiplikation verwenden wir drei verschachtelte Schleifen. Die beiden äußeren Schleifen wählen die zu multiplizierende Zeile bzw. Spalte aus, die innere summiert die Elementprodukte dieser Auswahl.

Programm:

Quellcode 13.2: matmul.py: Matrizenmultiplikation

```
from read import Read

def readMatrix(rows, cols):
    """
    Matrix mit rows Zeilen und cols Spalten
    zeilenweise einlesen.
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```

:~return: erzeugte Matrix
"""
a = [[0 for _ in range(cols)] for _ in range(rows)]
for r in range(rows):
    for c in range(cols):
        a[r][c] = Read.nextInt()
return a

def showMatrix(a):
    """
    Matrix ausgeben.
    Zeilen- und Spaltenzahl werden automatisch ermittelt.
    """
    for row in a: # row enthält Matrixzeile als Liste
        for val in row:
            print(" {:3d}".format(val), end='')
        print()

def matMul(a, b):
    """
    Matrixmultiplikation.
    Spaltenzahl von A muss mit Zeilenzahl von B übereinstimmen.
    :return: Produktmatrix oder None bei Fehler
    """
    rowsA = len(a)
    colsA = len(a[0])
    rowsB = len(b)
    colsB = len(b[0])
    if colsA != rowsB:
        # Falsche Dimension
        return None
    # Zielmatrix erzeugen Dimension rowsA x colsB
    res = [[0 for _ in range(colsB)] for _ in range(rowsA)]
    for r in range(rowsA):
        for c in range(colsB):
            sum = 0
            for i in range(colsA):
                sum += a[r][i]*b[i][c]
            res[r][c] = sum
    return res

def run():
    # erste Matrix lesen
    rows = Read.nextInt()
    cols = Read.nextInt()
    a = readMatrix(rows, cols)
    print("A:")
    showMatrix(a)
    # zweite Matrix lesen
    rows = Read.nextInt()
    cols = Read.nextInt()
    b = readMatrix(rows, cols)
    print("B:")
    showMatrix(b)
    # Multiplikation
    c = matMul(a, b)
    
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
print("AB:")
showMatrix(c)

if __name__ == '__main__':
    run()
```

Download: `matmul.py`.

Ergebnis für obiges Beispiel:

```
A:
  1  2  3  4
  5  6  7  8
  9 10 11 12

B:
  5  7
  1  3
  2  8
  9  1

AB:
 49 41
117 117
185 193
```

Wollen wir reellwertige Matrizen verarbeiten, müssen wir lediglich die Methode `nextInt()` durch `nextFloat()` ersetzen.

Bemerkung: Wollen wir umfangreiche mathematische Matrizenrechnungen durchführen, werden wir feststellen, dass unsere mehrdimensionalen Felder recht langsam verarbeitet werden. Hier bietet es sich an, auf spezielle Bibliotheken wie `numpy` zurückzugreifen, die eine geeignetere und trotzdem flexible Implementierung mehrdimensionaler Felder samt entsprechenden Operationen bereitstellen.

13.5 Verknüpfen von Listen

Um alle Elemente einer Liste zu verarbeiten, stehen uns zunächst zwei Möglichkeiten zur Verfügung:

1. über den Index:

```
for i in range(len(liste)):
    # verarbeite liste[i]
    # ...
```

2. ohne Index:

```
for element in liste:
    # verarbeite element
    # ...
```

Die zweite Form ist die leichter lesbare, stellt uns aber keinen Elementindex zur Verfügung. Wir können die durch eine zusätzliche Indexvariable lösen die wir parallel mitführen und aktualisieren:

```
idx = 0
for element in liste:
    # verarbeite idx, element
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
# ...
idx += 1
```

Statt dieser umständlichen Konstruktion stellt uns Python die Funktion `enumerate()` zur Verfügung, die aus jedem Listenelement ein Paar aus Index und Element erzeugt:

```
for idx, element in enumerate(liste):
    # verarbeite idx, element
    # ...
```

Beispiel: Auswahl der Monatsnummern aller »langen« Monate (mit 31 Tagen).

```
def idxLongMonths(months):
    res = list()
    for idx, mo in enumerate(months):
        if mo == 31:
            res.append(idx+1)
    return res
```

Verwendung:

```
monthLengths = [31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31]
longMonths = idxLongMonths(months) # → [1, 3, 5, 7, 8, 10, 12]
```

Ebenso ist häufig erforderlich, zwei gleichlange Listen elementweise parallel zu verarbeiten. Wir können hier wieder den Indexzugriff nutzen:

```
i = 0
for i in range(len(liste1)):
    # verknüpfe liste1[i] und liste2[i]
    # ...
```

Meist benötigen wir den Index aber nicht, sondern benötigen nur die jeweiligen Elementpaare. Dies erreichen wir mit der Funktion `zip()`.

```
for el1, el2 in zip(liste1, liste2):
    # verknüpfe el1, el2
    # ...
```

Beispiel: Skalarprodukt zweier Vektoren.

```
def dotproduct(vector1, vector2):
    # Test gleicher Länge
    if len(vector1) != len(vector2):
        return None
    sum = 0
    for el1, el2 in zip(vector1, vector2):
        sum += el1 * el2
    return sum
```

Verwendung:

```
v = [1, 3, 7, 4, -1]
# Quadrat der Norm
norm2 = dotproduct(v, v) # → 76
```

13.6 Aufgaben

Aufgabe 13.1 *Neuimplementierung C-Arrays*

Implementieren Sie die Probleme unter *Aufgaben (C-Felder)* nun in Python.

Aufgabe 13.2 *Spezielle quadratische Matrix*

Erzeugen Sie quadratische Felder nach folgender Bildungsvorschrift:

- Ein Feld der Größe 1×1 enthält die 1.
- Um aus einem Feld der Größe $k \times k$ ein Feld der Größe $(k+1) \times (k+1)$ zu erzeugen, wählen wir die nächstgrößere Zahl, die in $k \times k$ -Feld nicht vorkommt, und beginnen mit dieser zunächst rechts von oben nach unten mit fortlaufenden Werten die Spalte aufzufüllen. Wir gehen weiter bis zur Diagonale und von hier in der $k+1$ -ten Zeile wieder nach links.

Beispiel:

```
k = 1:
1

k = 2:
1 2
4 3

k = 3:
1 2 5
4 3 6
9 8 7
```

Erzeugen Sie eine solche Matrix für gegebene Dimension.

Aufgabe 13.3 *IBAN*

Prüfen Sie die Korrektheit einer *IBAN* (*Anleitung*). Lesen Sie die IBAN ziffernweise ein und ersetzen Sie in der Eingabe die Buchstabenkennung DE durch die beiden Zahlen 13 und 14, Beispiel: DE41500105170123456789 lesen wir als 13 14 4 1 5 0 0 1 0 5 1 7 0 1 2 3 4 5 6 7 8 9.

Aufgabe 13.4 *Geschenkverteilung*

In einem Dorf mit nur einer langen Straße soll der Weihnachtsmann allen Kindern Geschenke bringen. Da die Zahl der Geschenke aber so groß ist, muss er für jedes Kind einzeln zu seinem Lager fahren, die Geschenke zum Kind bringen und zum Lager zurückfahren. Nun möchte er das Lager (an der Straße) an der Stelle errichten, dass der Gesamtweg möglichst kurz wird.

Die Adressen der Kinder sind als Hausnummer (natürliche Zahl) gegeben, der Abstand zwischen zwei Hausnummern ist immer gleich.

Das Lager darf sich dabei

- auch an der Adresse eines Kindes befinden (Variante 1)
- *nicht* mit der Adresse eines Kindes zusammenfallen (Variante 2).

Aufgabe 13.5 *Sortieren von Eierkuchen*

Wir wollen einen Stoß von Eierkuchen mit verschiedenen Durchmessern sortieren. Unsere bisherigen Sortierverfahren funktionieren hier nicht, da wir keinen einzelnen Eierkuchen aus dem Stoß herauslösen können. Wir können lediglich mit einer Kuchenschaufel zwischen zwei Eierkuchen in den Stoß hineinstoßen, den kompletten oberen Stoß abheben und umgedreht wieder auf dem unteren Stoß ablegen.

Entwickeln Sie ein Sortierverfahren mit nur diesen erlaubten Operationen. Hinweise: [Wikipedia](#).

- *Einfache Stringoperationen*
- *Zeichencodes*
- *Stringlisten*
- *Wichtige Stringmethoden*
- *Aufgaben*

Im Gegensatz zu C besitzt Python einen echten String-Typ, der mit dem kompletten Unicode-Zeichensatz umgehen kann. Stringobjekte sind dabei unveränderlich, eine Stringbearbeitung erzeugt also stets eine neues String-Objekt.

Im Gegensatz zu Python 2 sind Strings in Python 3 stets Unicode-Strings, wir beschränken uns auf diese.

14.1 Einfache Stringoperationen

Stringliterals werden in Python ein einfache Apostrophe oder doppelte Anführungszeichen eingeschlossen, wobei die Begrenzer am Anfang und Ende des Literals übereinstimmen müssen:

```
'string'  
"string"
```

Im Gegensatz zur Unix-Shell oder zur Programmiersprache *PHP* gibt es keinen Unterschied bei der Interpretation der Begrenzer.

Warnung: Es dürfen nur die *ASCII*-Zeichen ' bzw. ", nicht aber die »typographisch korrekten« deutschen Anführungszeichen " bzw. ‘ statt des einfachen Apostrophs verwendet werden!

Soll das Begrenzungszeichen innerhalb des Strings verwendet werden, ist es mit einem Backslash zu schützen:

```
linkElement = "<a href=\"https://www.tu-chemnitz.de\">Homepage</a>"
zitat = 'Er sagte: "Shakespeare sagt: \'2 b or not 2 b\'"'
```

Benötigen wir Unicode-Zeichen im String, die wir nicht auf der Tastatur finden, können wir diese über ihren Codepunkt spezifizieren:

`\ucodepunkt`

Beispiele:

```
print('I \u2665 TU Chemnitz \u263b')
print('\u2600 und \u2602')
print('a \u2208 A \u2282 B')
```

Ergebnis:

```
I  TU Chemnitz
   und
a  A  B
```

Mehrzeilige Strings werden in dreifach wiederholte Apostrophe oder Anführungszeichen eingeschlossen:

```
'''
stringInhalt
...
'''
```

Werden nach dem beginnenden und vor dem beendeten Begrenzer Zeilenschaltungen eingefügt, sind diese im String enthalten. Wir haben diese Form schon als Dokumentationsstring von Python-Skripts und -Funktionen gesehen.

Um mit Strings zu arbeiten, weisen wir sie einer Variable zu.

```
variable = stringAusdruck
```

Die Länge eines Stringobjekts erhalten wir mit der Funktion `len()`, mit dem Indexoperator `:samp:{string}[{index}]` wählen wir einzelne Zeichen aus, die wiederum vom Typ String sind. Ebenso ist durch die Angabe eines Intervalls `string[startIndex:endIndex]` die Auswahl eines Teilstrings möglich. Dieser ist ein neues Objekt, der Originalstring ist weiter unverändert vorhanden.

```
place = 'Děčín'
for i in range(len(place)):
    letter = place[i]
    print(letter)

# oder besser
for letter in place:
    print(letter)
```

Bemerkung: Es ist nicht möglich, per Indexzugriff einzelne Zeichen des Strings auszutauschen, da Stringobjekte unveränderlich sind.

Stringobjekte können mit der Funktion `input()` eingelesen werden:

```
variable = input( [ 'Anzeige' ] )
```

Wir können optional einen Stringparameter angeben, der vor der Eingabe angezeigt wird. Es wird eine komplette Zeile gelesen, das Newline-Zeichen zum Abschluss der Eingabe ist nicht Bestandteil des Strings.

Wir können Strings zwar nicht verändern, aber aus vorhandenen Strings neue Strings zusammensetzen. Dazu stehen uns die Operatoren `+` für das Zusammenhängen und `*` für die Vervielfachung zur Verfügung.

Beispiel:

```
greeting = 'Ho'
desc = 'Ich bin der'
obj = 'Weihnachtsmann'

satz = (greeting + ' ')*3 + '... ' + desc + ' ' + obj
print(satz)
```

Ergebnis: An 'Ho' wird ein Leerzeichen angehängt, dann werden der String 3-mal wiederholt, eine Ellipse mit Leerzeichen und die beide Satzteile, durch Leerzeichen getrennt angehängt:

```
Ho Ho Ho ... Ich bin der Weihnachtsmann
```

Um einen String umzudrehen, kann man wie folgt vorgehen: Man schneidet das erste Zeichen ab, dreht den Reststring um und hängt das abgeschnittene Zeichen am Ende an:

```
def revStr(s):
    '''
    String umdrehen
    '''
    if len(s) == 0:
        return ''
    return revStr(s[1:]) + s[0]
```

Der Vergleich von zwei Strings ist direkt mit den üblichen Vergleichsoperatoren == und != möglich, wobei ein zeichenweiser Vergleich der Codepunkte dieser Zeichen durchgeführt wird, Groß- und Kleinschreibung wird also unterschieden. Ebenso stehen die Ordnungsoperationen <, <=, > und >= zur Verfügung.

14.2 Zeichencodes

Im Gegensatz zu C sind die einzelnen Zeichen eines Strings keine einfachen Zahlen, sondern wiederum Strings der Länge 1. Benötigen wir den Codepunkt (also Zahlenwert) eines Zeichens, müssen wir diesen mit der Funktion `ord()` ermitteln.

```
code = ord(zeichen)
```

Ein Zeichen ist hier ein String der Länge 1. Umgedreht kann auch ein Zeichen aus seinem Codepunkt erzeugt werden:

```
zeichen = chr(code)
```

Wir erhalten einen String der Länge 1. Sehr häufig ist es hier sinnvoll, den Codepunkt als Hexadezimalwert anzugeben (durch Voranstellen von `0x`):

```
char = chr(0x2665)
print(char, len(char))
```

Beispiel: Erzeuge einen neuen String, in dem jedes Zeichen durch das Zeichen mit dem nächstfolgenden Codepunkt ersetzt ist.

```
place = 'Kláštorec nad Ohří'
target = ''
for letter in place:
    code = ord(letter)
    newChar = chr(code + 1)
    target += newChar
# target == 'LmãŤufsfđ!obe!PiŠî'
```

14.3 Stringlisten

Da eine Python-Liste beliebige Objekte enthalten darf, können wir mit ihr auch Stringfelder realisieren:

```
stringListe = [ string0, string1, ... ]
```

Der Zugriff auf ein Listenelement liefert uns einen String. Beispiel:

```
greek = ['alpha', 'beta', 'gamma', 'delta']

print(greek[1])    # beta
print(greek[1:3])  # beta, gamma
print(greek[-1])   # gamma
```

Wollen wir auf ein Zeichen oder einen Teilstring eines Listenelements zugreifen, dann hängen wir einfach wie bei zweidimensionalen Feldern den zweiten Index an:

```
print(greek[-1][1]) # a (zweites Zeichen in 'gamma')
# Alle Anfangsbuchstaben
for i in range(len(greek)):
    print(greek[i][0])
# Achtung: So funktioniert es nicht:
print(greek[:][0]) # alpha (greek[:] liefert Liste, [0] wählt erstes Element)
```

14.4 Wichtige Stringmethoden

Die meisten Funktionen, die wir auf Strings anwenden, sind als Methoden (s. *Python-Klassen*) implementiert, wir hängen die Funktion also durch einen Punkt getrennt an unseren String bzw. die Stringvariable an:

```
stringObjekt.methode( parameterListe )
```

Sehr häufig enthält ein String, bedingt durch das Einlesen von der Konsole oder aus einer Datei am Anfang und Ende Leerzeichen oder Zeilenschaltungen, die bei der weiteren Verarbeitung stören. Diese können wir einfach mit der Methode `strip()` entfernen:

```
stringObjekt.strip()
```

Lesen wir Strings aus einer Datei zeilenweise ein, stehen wir oft vor der Aufgabe, diese Zeile an Trennzeichen in ihre Bestandteile zu zerlegen. Dies erledigt die Methode `split()`:

```
stringListe = stringObjekt.split( [trennString] )
```

Ohne die Angabe des Trennstrings wird der String an allen »white spaces« (Leerzeichen, Tabulator, Zeilenschaltung) zerlegt, wobei wiederholte Trennzeichen keine leeren Strings in der Stringliste erzeugen. Wird dagegen ein Trennstring angegeben, führt dessen Wiederholung zu leeren Strings im Ergebnis:

```
s = ' a  b ' # Leerzeichen a 3 Leerzeichen b Leerzeichen
s.split()    # -> ['a', 'b']
s.split(' ') # -> ['', 'a', '', '', 'b', '']

t = '3, 4, 5'
t.split(',') # -> ['3', '4', '5']
```

Das Ergebnis ist stets eine Liste von Strings, auch wenn die Bestandteile scheinbar Zahlen sind. Wollen wir diese im weiteren als Zahlen verarbeiten, dann müssen wir sie konvertieren:

```
intVariable = int(intString)
floatVariable = float(intString)
boolVariable = bool(boolString)
```

Beispiel: Zerlegen einer Liste von Städten, wobei jeder Eintrag aus den Komponenten Name, Einwohnerzahl, Längen- und Breitengrad besteht und die Werte durch ; getrennt sind.

Quellcode 14.1: readplaces.py: Städteliste lesen und verarbeiten

```
def run():
    '''
    Städte zeilenweise lesen bis Zeile ###.
    Einwohner summieren.
    Mittlere Länge und Breite bestimmen
    Ortsliste, durch Komma getrennt, erzeugen
    '''
    line = input()
    sumPers = 0
    cnt = 0
    sumLon = 0.0
    sumLat = 0.0
    places = list()
    while line != '###':
        parts = line.split(';')
        # Debug
        cnt += 1
        print(parts)
        places.append(parts[0])
        pers = int(parts[1])
        sumPers += pers
        lon = float(parts[2])
        sumLon += lon
        lat = float(parts[3])
        sumLat += lat
        # nächste Zeile lesen
        line = input()
    # Auswertung
    places = ', '.join(places)
    print('Orte: {}'.format(places))
    print('Gesamteinwohner: {}'.format(sumPers))
    print('Mittlere Position: {:.2f} n. B. {:.2f} ö. L.'.format(
        sumLat/cnt, sumLon/cnt))

if __name__ == '__main__':
    run()
```

Download: [readplaces.py](#).

Beispieldaten:

```
Chemnitz;250000;12.9;50.8
Dresden;560000;13.7;51.0
Leipzig;593000;12.4;51.3
###
```

Download: [places.dat](#).

Ausgabe:

```
['Chemnitz', '250000', '12.9', '50.8']
['Dresden', '560000', '13.7', '51.0']
['Leipzig', '593000', '12.4', '51.3']
Orte: Chemnitz, Dresden, Leipzig
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
Gesamteinwohner: 1403000
Mittlere Position: 51.03 n. B. 13.00 ö. L.
```

Umgedreht kann es oft sinnvoll, Stringlisten wieder zu einem einzelnen String zusammenzusetzen. Dazu dient die Stringmethode `join()`:

```
lueckenString.join(stringListe)
```

Der Lückenstring gibt das Verbindungszeichen an, das zwischen die benachbarten Listenelemente eingefügt werden soll. Beispiel:

```
greek = ['alpha', 'beta', 'gamma', 'delta']
print(' '.join(greek))
print(' - '.join(greek[1:3]))
print(''.join(greek[:-1]))
```

Ergebnis:

```
alpha beta gamma delta
beta - gamma
alphabetagamma
```

Das Zusammenfügen funktioniert so jedoch nur für Strings, wollen wir Listen anderer Datentypen zu einem String vereinigen, müssen die Elemente vorher in Strings konvertiert werden. Ein einzelnes Objekt kann mit der Funktion `str()` umgewandelt werden (für eigene Typen muss diese Funktion passend definiert werden, siehe dazu im Kapitel *Python-Klassen*):

```
stringObj = str(ausdruck)
```

Zum Erzeugen einer Stringliste aus einer Liste anderer Objekte können wir wieder die Generatorfunktionalität verwenden:

```
stringListe = [str(val) for val in liste]
```

Beispiel: String aus Zahlenliste erzeugen, wobei die Zahlen durch Komma und Leerzeichen voneinander getrennt sind.

```
numbers = [1, 2, 3, 4, 5]
# Erzeugen einer Stringliste aus der Zahlenliste und Verkettung dieser Liste
numberStr = ', '.join([str(x) for x in numbers])
```

Um einen String komplett in Großbuchstaben umzuwandeln, können wir die Methode `upper()` verwenden. Diese Methode kann auch korrekt mit Umlauten und akzentuierten Buchstaben umgehen.

```
s = 'Plätzchen, Weißwein, Palačinka s šlehačkou, '
print(s.upper()) # → PLÁTZCHEN, WEISSWEIN, PALAČINKA S ŠLEHAČKOU,
```

14.5 Aufgaben

Aufgabe 14.1 Cäsar-Chiffre

Schreiben Sie ein Programm, das einen Text nach der Cäsar-Chiffre verschlüsselt: Jeder (*ASCII*-)Buchstabe soll durch einen Buchstaben ersetzt werden, der k Positionen rechts von ihm steht. Wird das *z* erreicht, beginnt die Zählung wieder bei *a*. Dabei sollen Großbuchstaben und Großbuchstaben, Kleinbuchstaben in Kleinbuchstaben und Ziffern in Ziffern umcodiert werden. Alle anderen Zeichen bleiben unverändert.

Schreiben Sie zur Kontrolle auch eine Entschlüsselung, die eine analoge Verschiebung um k Zeichen nach links durchführt.

Aufgabe 14.2 *Palindrom*

Lesen Sie eine Liste von Strings per Dateiumlenkung zeilenweise ein und überprüfen Sie für jeden String, ob er ein Palindrom ist, er ergibt also von vorn und hinten gelesen den gleichen Inhalt, wobei Satz- und Leerzeichen sowie Groß- und Kleinschreibung unberücksichtigt bleiben.

Beispiele:

```
Lagerregal
Otto
Anna
Eine treue Familie bei Lima feuerte nie
```

Aufgabe 14.3 *Anagramm*

Lesen Sie aus einer Datei Stringpaare, die Sie durch einen senkrechten Strich (|) trennen, zeilenweise ein und prüfen Sie, ob die Strings Anagramme sind, ob sich also der zweite String durch Umordnen der Buchstaben des ersten Strings ergibt, ohne Zeichen hinzuzufügen oder wegzulassen. Leerzeichen, Satzzeichen und Groß-/Kleinschreibung bleiben dabei unberücksichtigt.

Beispiele:

```
Angela Merkel|Karamelengel
Poenisch|Scheinpo
Ida, melke Opa!Kamelopedia
```


- *Grundlagen*
- *Klassendefinition*
- *Methoden*
- *Zugriff auf Komponenten (Getter und Setter)*
- *Objekte und Funktionen*
- *Objektlisten*
- *Aufgaben*

15.1 Grundlagen

Wir haben in C mit dem `struct`-Konstrukt eine Möglichkeit kennengelernt, zusammengehörige Daten zu einer Einheit zusammenzufassen. Oft benötigen wir speziell auf diese Daten zugeschnittene Funktionen, die entweder nur für diese Daten sinnvoll sind (wie das Kürzen beim Datentyp Bruch oder das Anhängen von Elementen an eine Liste) oder je nach Datentyp unterschiedlich implementiert werden müssen (wie die Ausgabe eines Bruchs oder einer Liste).

Die *OOP* erweitert das Konzept der Datenkapselung auf die zu diesen Daten zugehörigen Funktionen, die nun *Methoden* genannt werden. Weiter ermöglicht es sie, bestimmte Methoden für jeden Datentyp unterschiedlich zu implementieren (*Polymorphie*). Ein drittes Konzept ist die *Vererbung*, die es erlaubt, aus vorhandenen Datentypen neue Datentypen abzuleiten, wobei die bisherigen Daten und Funktionalität übernommen (geerbt) werden und neue Funktionen und Daten ergänzt werden können.

Wir beschränken uns an dieser Stelle im Wesentlichen auf das Konzept der *Kapselung* und gehen nur kurz auf die *Polymorphie* ein.

Einen eigenen Struct-Typ kennt Python nicht, wir müssen also das Klassenkonzept verwenden. Eine Alternative, die sich auf die reine Kapselung der *Daten* beschränkt, stellen die *Dictionaries* dar, die wir hier nicht besprechen wollen.

15.2 Klassendefinition

Wir müssen zunächst den Aufbau der Klasse als neuen Datentyp definieren, ehe wir Objekte dieser Klasse anlegen können:

```
class KlassenName:
    def __init__(self, parameterliste):
        self.komponente = ausdruck
        ...

    [
        def methode(self, parameterliste):
            ...
        ...
    ]
```

Der Klassenname sollte groß geschrieben werden. Beim Erzeugen eines Klassenobjektes wird die `__init__()`-Methode aufgerufen (Schreibweise: zwei Unterstriche vor und nach dem Wort `init`), mit der die Komponenten des Objekts angelegt und initialisiert werden. Der erste Parameter `self` referenziert das gerade zu erzeugende Objekt, die Komponenten werden wie bei C-Structs durch einen Punkt abgetrennt an `self` angehängt.

Die Definition eines Bruchs (s. *Structs*) könnte so erfolgen. Wie bei jeder Python-Funktion können wir Parameter mit vordefinierten Werten belegen und diese dann beim Aufruf weglassen.

```
class Fraction:
    def __init__(self, nom, denom = 1):
        self.nom = nom
        self.denom = denom
        if self.denom < 0:
            self.nom = -self.nom
            self.denom = -self.denom
        self.val = self.nom / self.denom # in Python3 Gleitpunktoperator
```

Diese Definition ist der »Bauplan« mit dem wir *Objekte* oder *Instanzen* erzeugen können. Wir benutzen dazu den Klassennamen wie einen Funktionsruf. Eventuell Parameter hinter dem Klassennamen werden der Methode `__init__()` übergeben. Diese fügt den Parameter `self` automatisch hinzu, mit dem das gerade erzeugte Objekt angesprochen (referenziert) wird:

```
third = Fraction(1, 3)
two = Fraction(2)
```

Jedes dieser Objekte hat die Komponenten `nom`, `denom` und `val`, auf die wir mit der gleichen Syntax wie bei Structs in C zugreifen:

```
print(third.nom, third.denom)
# val neu berechnen
two.val = two.nom / two.denom
```

Python-Klassen stellen also zunächst das Äquivalent zu C-Structs dar. Die Klassendefinition inklusive der Methode `__init__()` ersetzen die C-Typdefinition des Structs. Ein Objekt bzw. eine Instanz dieser Klasse ersetzen die Definition der Struct-Variable, wobei die Erzeugung über einen Konstruktor erfolgt und sicherstellt, dass das Objekt initialisiert ist.

15.3 Methoden

Im Gegensatz zu Structs können wir unseren Klassen zusätzlich Methoden hinzufügen, die speziell auf Objekte dieser Klassen zugeschnitten sind. Diese Zusammenfassung von zusammengehörigen Daten und Funktionalität eines Objekts zu einer Einheit bezeichnet man als *Kapselung*.

Eine Methode wird wie eine normale Python-Funktion definiert mit folgenden Unterschieden:

- sie wird innerhalb einer Klassendefinition definiert (also entsprechend eingerückt)
- Der erste Parameter, meist `self` genannt, ist eine Referenz auf das Objekt, für das die Methode gerufen wird.

Damit ergibt sich folgende Syntax:

```
class Klasse ... :
    ...
    def methode(self [, parameterListe ]):
        funktionsBlock
```

Mit der Referenz `self` können wir, durch einen Punkt abgetrennt, auf die Komponenten und weitere Methoden des Objekts zugreifen, wie wir dies schon beim Konstruktor `__init__()` gesehen haben.

Beispiel: Kürzen eines Bruchs. Wir verwenden eine Funktion `gcd()`, die wir entweder selbst mit dem Euklidischen Algorithmus implementieren oder dem Python-Modul `math` entnehmen.

```
class Fraction:
    # __init__() wie oben
    # ...

    def shorten(self):
        div = gcd(self.nom, self.denom)
        if div > 1:
            self.nom //= div
            self.denom //= div
        # self.val muss nicht korrigiert werden

# Aufrufbeispiel:
f1 = Fraction(4, 6)
f1.shorten()
print(f1.nom, f1.denom)
```

Wollen wir ein Objekt mit `print()` ausgeben, erhalten wir eine Anzeige der Form:

```
<__main__.Fraction object at 0x7f16946d3c10>
```

Wir sehen den Klassennamen sowie eine Objekt-Id, die etwa der Speicheradresse einer mit `&` dereferenzierten C-Variablen entspricht (hier nicht behandelt). Diese Ausgabe können wir verbessern, indem wir für unsere Klasse die Methode `__str__()` (je 2 Unterstriche vor und nach `str`) definieren (*überschreiben*). Sie erwartet außer `self` keine Parameter und muss einen String zurückgeben:

```
def __str__(self):
    return "{}/{ } (= {:.2f})".format (self.nom, self.denom, self.val)
```

Wir können nun Bruch-Objekte einfach mit

```
print(f2)
```

ausgeben und erhalten eine sinnvolle Anzeige:

```
2/3 (= 0.67)
```

Zur Demonstration einer Methode, die einen Parameter besitzt und einen Wert zurückliefert, wollen wir die Multiplikation zweier Brüche implementieren:

```
class Fraction:
    # ...
    def mul(self, fraction):
        """
        Multipliziert den Bruch mit dem übergebenen Bruch und
        erzeugt einen neuen Bruch als Ergebnis.
        Das Objekt selbst bleibt unverändert.
        """
        return Fraction(self.nom * fraction.nom, self.denom * fraction.denom)
```

Aufruf:

```
f1 = Fraction(3, 5)
f2 = Fraction(2, 3)
res = f1.mul(f2)
print(res)
```

15.4 Zugriff auf Komponenten (Getter und Setter)

Der direkte Zugriff auf die Komponenten sollte vermieden werden, statt dessen erfolgt dieser über spezielle *Getter*- und *Setter*-Methoden. Der Vorteil dieser Vorgehensweise ist, dass wir zum einen fehlerhafte Werte beim Setzen abfangen können, zum anderen können wir ohne Auswirkungen auf das restliche Programm die Implementierung der Klasse ändern, solange die *Schnittstelle* (i. e. die Signaturen aller Methoden) gleich bleibt.

Python realisiert diesen Mechanismus normalerweise mit *Properties*, wir wollen hier jedoch den Java-Ansatz wählen, der leichter zu verstehen ist. Dazu verwenden wir für die Komponenten unserer Klasse Namen, die mit zwei Unterstrichen beginnen:

```
class Klasse:
    ...
    def __init__(self, parameterListe):
        __komponente = ausdruck
    ...
```

Eine solche Komponente ist nun privat, i. e. sie kann außerhalb der Klasse nicht mehr angesprochen werden. Statt dessen schreiben wir zwei Methoden

```
def getKomponente(self):
    return self.__komponente

def setKompente(self, wert):
    self.__komponente = wert
```

Statt der einfachen Zuweisung in beiden Methoden natürlich beliebige Algorithmen zum Testen bzw. zur Transformation der Parameter implementiert werden.

Soll das Objekt nach dem Anlegen komplett unveränderlich sein, lassen wir die Setter-Methoden weg. Ebenso können wir das Ändern einzelner Komponenten verbieten, indem wir nur für diese Komponente keine Setter-Methode bereitstellen.

Beispiel unserer Bruch-Klasse. Nur der Zähler soll nachträglich verändert werden können.

```
class Fraction:

    def __init__(self, nom, denom):
        self.__nom = nom
        self.__denom = denom
        if self.__denom < 0:
            self.__nom = -self.__nom
            self.__denom = -self.__denom
        self.__val = self.__nom / self.__denom

    def getNom(self):
        return self.__nom

    def getDenom(self):
        return self.__denom

    def getVal(self):
        return self.__val

    def setNom(self, nom):
        self.__nom = nom
        # Float-Wert muss korrigiert werden
        self.__val = self.__nom / self.__denom

    # weitere Methoden
    # ...
```

Versuchen wir, auf die Komponenten lesend zuzugreifen, erhalten wir eine Fehlermeldung:

```
print(x2, x2.__nom, x2.__denom)
# → AttributeError: 'Fraction' object has no attribute '__nom'
```

Versuchen wir eine Komponente direkt zu ändern, scheint auf den ersten Blick zu funktionieren – wir erhalten keine Fehlermeldung – aber der Wert bleibt unverändert:

```
x1 = Fraction(2, 3)
print(x1)           # → 2/3 (= 0.67)
x1.__nom == 5
print(x1)           # → 2/3 (= 0.67)
```

Statt dessen müssen wir die Komponenten mittels Getter- und Setter-Methoden ansprechen:

```
print(x1, x1.getNom(), x1.getDenom(), x1.getVal())
x1.setNom(23)
print(x1)
```

Download des kompletten Beispielcodes für die Bruchklasse: `fraction.py`.

15.5 Objekte und Funktionen

Übergeben wir Objekte an eine Funktion, erfolgt die Übergabe wie bei auch bei einfachen Datentypen stets als *Referenz* auf das beim Aufruf angegebene Objekt. Während aber Objekte einfacher Datentypen und auch Strings unveränderlich sind, verhält sich deren versuchte Veränderung innerhalb der Funktion wie eine Übergabe als Wert: es wird ein neues Objekt mit dem neuen Wert erzeugt, das übergebene Objekt bleibt unverändert.

Unsere selbstdefinierten Objekte sind dagegen veränderbar. Werden sie als Parameter an eine Funktion übergeben und dort verändert, ist damit das Originalobjekt verändert.

Beispiel: Multiplikation eines Bruchs mit einer Zahl als Funktion.

```
def scaleFraction(frac, factor):
    """
    Skalieren des Bruch-Objekts mit dem Faktor.
    Das Bruch-Objekt wird verändert.
    """
    frac.setNom(frac.getNom() * factor)

# Aufruf
f1 = Fraction(2, 3) # → 2/3 (= 0.67)
scaleFraction(f1, 2)
print(f1)           # → 4/3 (= 1.33)
```

Geschickter ist es hier jedoch, die Skalierung als Methode zu implementieren:

```
class Fraction:

    # ... (siehe oben)

    def scale(self, factor):
        """
        Skalieren des Bruch-Objekts mit dem Faktor.
        Das Bruch-Objekt wird verändert.
        """
        self.__nom *= factor
        self.__val *= factor
        # oder Benutzung des Setters:
        # self.setNom(self.getNom() * factor)

# Aufruf
f1 = Fraction(2, 3) # → 2/3 (= 0.67)
f1.scale(2)
print(f1)           # → 4/3 (= 1.33)
```

Wollen wir die Veränderung des Originalobjekts vermeiden, müssen wir manuell von diesem eine Kopie erstellen, die wir dann manipulieren. Viele Programmiersprachen lösen diese Aufgabe mit einem *Copy-Constructor*. In Python können wir für diesen Zweck eine Methode `copy()` schreiben, die uns das Objekt klonet:

```
class Fraction:
    # ...
    def copy(self):
        """
        Kopie der Instanz erzeugen
        """
        return Fraction(self.__nom, self.__denom)
```

Damit können wir nun eine neue Methode schreiben, die aus einem Bruch, der unverändert bleiben soll, einen neuen, umskalierten Bruch erzeugt und diesen zurückgibt:

```
class Fraction:
    # ...
    def scaleCopy(self, factor):
        """
        Erzeugt neuen, skalierten Bruch.
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```

"""
res = self.copy()           # Objekt kopieren
res.setNom(self.__nom * factor) # ... und Kopie verändern
return res                  # Rückgabe der Kopie

# Aufruf
f1 = fraction(2, 3)
f2 = f1.scaleCopy(2)
print(f1, f2) # → f1=2/3 (= 0.67), f2=4/3 (= 1.33)

```

15.6 Objektlisten

Wollen wir eine Menge gleichartiger Objekte verwalten, können wir wieder unseren Listentyp verwenden. Wir legen dazu zunächst eine leere Liste an und fügen neue Objekte mit `append()` hinzu.

```

fractions = list()
fractions.append(Fraction(1, 2))
fractions.append(Fraction(2, 3))
# ...

```

Wollen wir die Liste mit einer gewünschten Größe mit einem Default-Objekt anlegen, dann dürfen wir den `*`-Operator *nicht* verwenden, da die Liste dann Referenzen auf ein und dasselbe Objekt enthält:

```

fractions = [Fraction(0)]*10
fractions[0].setNom(1)      # → alle Brüche haben nun den Wert 1!

```

Statt dessen müssen wir unbedingt den Generatorausdruck verwenden, um tatsächlich unterschiedliche Objekte zu erzeugen:

```

fraction = [Fraction(0) for _ in range(10)]
fractions[0].setNom(1)      # → nur der 1. Bruch ist verändert

```

Beispiel: Verarbeitung der Städteliste `places.dat`. Definition der Klasse `Place`. Die `Place`-Objekte werden in einer Liste verwaltet. Diese soll angezeigt werden, in ihr soll der größte Wert bzw. der Mittelwert bzgl. einer bestimmten Komponente ermittelt werden. Weiterhin soll die Liste nach jeder Komponente auf- und absteigend sortiert werden können. Nach der Ortsliste lesen wir Namenspaare, suchen diese Orte in der Liste und bestimmen die Entfernung zwischen ihnen.

Quellcode 15.1: `places.py`: Verwaltung Ortsverzeichnis als Liste von Objekten

```

"""
Stadtliste auswerten.
Aufbau Daten:
Ort;Einwohner;Länge;Breite
...
###
Name1-Name2
...
###
"""

import math # für Abstand zwischen Orten

class Place:

```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```

"""
Ort, Komponenten:
name - Ortsname
size - Einwohnerzahl
lon  - Geogr. Länge
lat  - Geogr. Breite
"""
def __init__(self, name, size, lon, lat):
    self.__name = name
    self.__size = size
    self.__lon = lon
    self.__lat = lat

def __str__(self):
    """
    Umwandlung in String
    """
    return 'Name: {} ({} Einwohner) Lage: ({},{})'.format(
        self.__name, self.__size, self.__lon, self.__lat)

def __repr__(self):
    """
    Objektrepräsentation als String.
    Hier verkürzte Darstellung
    """
    return '<"{}", {}, {}, {}>'.format(
        self.__name, self.__size, self.__lon, self.__lat)

def getName(self):
    return self.__name

def getSize(self):
    return self.__size

def getLon(self):
    return self.__lon

def getLat(self):
    return self.__lat

def getField(self, field):
    """
    Auswahl der Komponente field.
    :param field: String für Feldauswahl
    """
    if field == 'name':
        return self.__name
    elif field == 'size':
        return self.__size
    elif field == 'lon':
        return self.__lon
    elif field == 'lat':
        return self.__lat

def compare(self, other, field):
    """

```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```

Vergleiche self mit other bzgl. field
:return: <0, wenn self < other; 0, wenn self = other, >0, wenn self > other
"""

v1 = self.getField(field)
v2 = other.getField(field)
if field == 'name':
    return compareStr(v1, v2)
else:
    return v1 - v2

def dist(self, other):
    """
    Entfernung zwischen zwei Orten (einfache Formel).
    siehe Orthodrome: https://de.wikipedia.org/wiki/Orthodrome
    """

    phiA = math.radians(self.__lat)
    phiB = math.radians(other.getLat())
    lambdaA = math.radians(self.__lon)
    lambdaB = math.radians(other.getLon())
    zeta = math.acos(math.sin(phiA)*math.sin(phiB) +
        math.cos(phiA)*math.cos(phiB)*math.cos(lambdaA-lambdaB))
    return zeta * 6371 # Erdradius in km

def compareStr(v1, v2):
    """
    Vergleich zweier String unabhängig von Groß-/Kleinschreibung.
    Achtung: Umlaute/ß/akzentuierte Buchstaben werden falsch sortiert.
    """

    if v1.upper() < v2.upper():
        return -1
    elif v1.upper() > v2.upper():
        return 1
    else:
        return 0

def readPlaces():
    """
    Lesen der Daten.
    :return: Liste von Place-Objekten
    """

    line = input()
    places = list()
    while line != '###':
        name, sSize, sLon, sLat = line.split(';')
        # Teile sind vom Typ str → konvertieren
        size = int(sSize)
        lon = float(sLon)
        lat = float(sLat)
        # Objekt erzeugen
        place = Place(name, size, lon, lat)
        # und an das Ergebnis anhängen
        places.append(place)
        line = input()
    return places

def showList(places):

```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```

for place in places:
    print(place) # verwendet Place.str()

def argMax(places, field):
    """
    Suche Position des größten Elements bzgl. field.
    Python kennt keinen enum-Typ, deshalb field als string
    """
    if len(places) == 0: # leere Liste
        return None
    iMax = 0
    valMax = places[0].getField(field)
    for i, place in enumerate(places):
        val = place.getField(field)
        if val > valMax:
            iMax = i
            valMax = val
    return iMax

def sortPlaces(places, field, direction=1):
    """
    Insertion Sort
    :param direction: 1 = aufsteigend, -1 = absteigend
    """
    size = len(places)
    for i in range(1, size):
        h = places[i]
        k = i
        while k > 0 and direction * places[k-1].compare(h, field) > 0:
            places[k] = places[k-1]
            k -= 1
        places[k] = h

def mean(places, field):
    if len(places) == 0:
        return 0.0
    sum = 0.0
    for place in places:
        sum += place.getField(field)
    return sum/len(places)

def readNamesPair():
    line = input()
    if line[0] == '#':
        return None, None
    return line.split('-')

def placeIdxOfName(places, name):
    """
    Suche Position von Ort in Liste places.
    :return: Listenindex oder -1, wenn nicht gefunden
    """
    for i, place in enumerate(places):
        if place.getName() == name:
            return i
    return -1
    
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
def showDist(places, name1, name2):
    idx1 = placeIdxOfName(places, name1)
    idx2 = placeIdxOfName(places, name2)
    if idx1 >= 0 and idx2 >= 0:
        dist = places[idx1].dist(places[idx2])
        print('Entfernung {}-{}: {:.0f} km'.format(name1, name2, dist))

def run():
    places = readPlaces()
    print("Direkte Listenausgabe:")
    print(places) # verwendet Place.repr()
    print("Eigene zeilenweise Listenausgabe:")
    showList(places)
    # nördlichste Stadt suchen
    idx = argMax(places, 'lat')
    print('Nördlichste Stadt: {}'.format(places[idx]))
    meanSize = mean(places, 'size')
    print('Mittlere Größe {:.1f}'.format(meanSize))
    meanLon = mean(places, 'lon')
    meanLat = mean(places, 'lat')
    print('Schwerpunkt (mittlere Lage): (:.2f{}, :.2f{})'.format(
        meanLon, meanLat))
    # Sortieren nach Größe absteigend
    sortPlaces(places, 'size', -1)
    print("Sortiert nach Größe:")
    showList(places)
    # Sortieren nach Name aufsteigend
    sortPlaces(places, 'name')
    print("Sortiert nach Name:")
    showList(places)
    # Lesen von Ortspaaren
    name1, name2 = readNamesPair()
    while name1 is not None:
        showDist(places, name1, name2)
        name1, name2 = readNamesPair()

if __name__ == '__main__':
    run()
```

Ergebnis:

```
Direkte Listenausgabe:
[<"Chemnitz", 250000, 12.9, 50.8>, <"Dresden", 560000, 13.7, 51.0>, <"Leipzig", 593000, 12.4, 51.3>, <"Berlin", 3700000, 13.4, 52.5>]
Eigene zeilenweise Listenausgabe:
Name: Chemnitz (250000 Einwohner) Lage: (12.9,50.8)
Name: Dresden (560000 Einwohner) Lage: (13.7,51.0)
Name: Leipzig (593000 Einwohner) Lage: (12.4,51.3)
Name: Berlin (3700000 Einwohner) Lage: (13.4,52.5)
Nördlichste Stadt: Name: Berlin (3700000 Einwohner) Lage: (13.4,52.5)
Mittlere Größe 1275750.0
Schwerpunkt (mittlere Lage): (:.2f13.1, :.2f51.4)
Sortiert nach Größe:
Name: Berlin (3700000 Einwohner) Lage: (13.4,52.5)
Name: Leipzig (593000 Einwohner) Lage: (12.4,51.3)
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
Name: Dresden (560000 Einwohner) Lage: (13.7,51.0)
Name: Chemnitz (250000 Einwohner) Lage: (12.9,50.8)
Sortiert nach Name:
Name: Berlin (3700000 Einwohner) Lage: (13.4,52.5)
Name: Chemnitz (250000 Einwohner) Lage: (12.9,50.8)
Name: Dresden (560000 Einwohner) Lage: (13.7,51.0)
Name: Leipzig (593000 Einwohner) Lage: (12.4,51.3)
Entfernung Chemnitz-Dresden: 60 km
Entfernung Chemnitz-Berlin: 192 km
```

15.7 Aufgaben

Aufgabe 15.1 Methode Kürzen

Implementieren Sie das Kürzen von Brüchen. Rufen Sie diese Methode direkt im Konstruktor auf.

Aufgabe 15.2 Tourenverwaltung, Hausaufgabe (8 Punkte)

Erstellen Sie ein Programm zur Verwaltung von Touren (Wanderungen, Rad- und Skitouren). Die Eingabedatei hat folgenden Aufbau:

- Zeile Tourenbeschreibung, bestehend aus Name, Startpunkt, Ziel, Länge in km, Typ (Fuß, Rad, Ski, ...), Wertung (1 ... 5). Die Felder sind durch Semikolon (;) getrennt.
- Mehrere Folgezeilen, die die Sehenswürdigkeiten der Tour beschreiben. Diese starten mit einem * und enthalten durch Semikolon (;) getrennt die Angaben Name der Sehenswürdigkeit und Entfernung vom Startpunkt.
- Nun folgt die nächste Tourenbeschreibung mit gleichem Aufbau.
- Die Datei wird durch die Zeile

```
###;;0;0
```

abgeschlossen (also Name ist ###, Start, Ziel, Typ sind leer, Länge und Bewertung sind 0).

- Es folgt eine Liste von einigen Namen der oben aufgeführten Touren, die genauer untersucht werden sollen. Jeder Name steht auf einer eigenen Zeile, die letzte Zeile enthält den Stopp-String ###.

Beispieldatei: `touren0.dat`.

Erstellen Sie geeignete Klassen zur Verwaltung der Touren und ihrer Sehenswürdigkeiten sowie geeignete Listen, um alle Touren- und Sehenswürdigkeiten zu verwalten.

Alle Komponenten der Klassen sollen privat sein, der Zugriff darf nur über Getter- bzw. Setter-Methoden erfolgen.

Lösen Sie folgende Teilaufgaben in Funktionen bzw. Methoden. Ihr Steuerprogramm soll jede dieser Teilaufgaben testen und eine geeignete Ausgabe erzeugen.

- Ausgabe einer Tourenliste, die neben allen Tourdaten auch alle Sehenswürdigkeiten dieser Tour auflistet.
- Erzeugen Sie eine nach Name sortierte Liste der Touren. Es soll nur die Anzahl der Sehenswürdigkeiten jeder Tour ausgegeben werden.
- Sortieren Sie die Tourenliste nach Tourtyp und innerhalb der Typs *absteigend* nach Länge. Bei gleicher Länge soll nach Name *aufsteigend* sortiert sein.

- Erzeugen Sie eine Liste aller Sehenswürdigkeiten, unabhängig von der Tour. Diese soll die Anzahl der Touren enthalten, die diese Sehenswürdigkeit aufsuchen (im Beispiel werden das Museum Gunzenhauser von 2 Touren, alle anderen nur von einer Tour besucht).
- Für die zu genauer zu untersuchenden Touren soll eine Liste der Tourabschnitte zwischen Start und erster Sehenswürdigkeit, jeweils benachbarten Sehenswürdigkeiten und der letzten Sehenswürdigkeit und dem Ziel mit den jeweiligen Abschnittslängen erzeugt werden. Für die Anzeige ist eine Ausgabefunktion zu realisieren, die diese Liste entgegennimmt und die Ausgabe beispielsweise in dieser Form realisiert (lange Texte dürfen in der Ausgabe abgeschnitten werden):

```
Tour: Kammtour (Bärenstein-Johanngeorgenstadt), 50 km
```

Abschnitt	Länge
-----	-----
Bärenstein→Klínovec	20
Klínovec→Anton-Günther-G	5
Anton-Günther-G→Plattenberg	15
Plattenberg→Eisbinge Horní	3
Eisbinge Horní →Johanngeorg	7

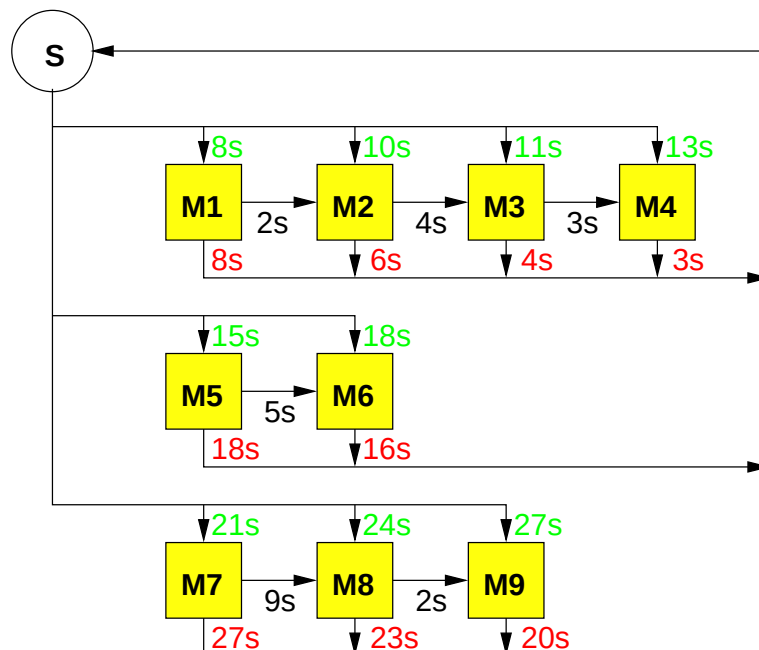
Das Programm ist in Python unter Verwendung von Klassen für die Beschreibung der Touren und Sehenswürdigkeiten zu realisieren. Bei Bedarf dürfen weitere Klassen definiert werden.

Die Verwendung der Bibliotheksfunktionen zum Sortieren ist hier nicht zulässig.

16.1 Ergänzende Aufgaben zu den behandelten Themen

Aufgabe 16.1 *Fabrikhalle*

In einer Werkhalle stehen verschiedene Maschinen, die Werkstücke in einer bestimmten Reihenfolge bearbeiten sollen. Die Werkstücke werden auf einer Startposition S angeliefert und dort nach der Bearbeitung wieder abgeholt. Jede Maschine kann von der Startposition direkt angefahren werden, ebenso kann das Werkstück nach der Bearbeitung an die Startposition zurückgebracht werden. Zwischen einigen Maschinen gibt es Direktverbindungen, die auch benutzt werden dürfen, wenn eine Maschine dazwischen übersprungen wird.



Zu den Wegzeiten kommt die Bearbeitungszeit auf der jeweiligen Maschine, die für jede Maschine ein fester Wert ist (z.B. M1: 12s).

- Definieren Sie geeignete Datenstrukturen zur Modellierung dieser Aufgabenstellung.
- Lesen Sie alle benötigten Daten aus einer Datei ein.
- Prüfen Sie die Korrektheit der Daten (jede Maschine kann von der Startposition erreicht werden und umgekehrt, Maschinen sind nur in einer Richtung untereinander verbunden, ...).
- Bestimmen Sie die Zeit (Weg- und Bearbeitungszeit) für ein Werkstück, das auf genau einer Maschine bearbeitet werden muss.
- Ermitteln Sie die Maschine, die die kürzeste Bearbeitungszeit ermöglicht (incl. Weg).
- Ermitteln Sie die reine Bearbeitungszeit (ohne Wegzeiten) für ein Werkstück auf einer gegebenen Folge von Maschinen.
- Geben Sie eine Bearbeitungsfolge vor und ermitteln Sie den Bearbeitungsweg zwischen den Maschinen (einschließlich des Rangierens über die Startposition) sowie die Gesamtbearbeitungszeit.
- Ermitteln Sie für eine Folge von Bearbeitungsschritten die günstigste und ungünstigste Reihenfolge, wobei in der Folge keine Maschine doppelt benutzt werden soll.
- Ermitteln Sie die längste mögliche Folge von Bearbeitungsschritten, ohne zwischendurch die Startposition anfahren zu müssen.

Stellen Sie die Maschinenaufstellung als ASCII-Graphik dar. Es sollen nur die Verbindungen zwischen den Maschinen dargestellt werden. Bei jeder Maschine soll die Anfahrtszeit aus der Startposition, die Bearbeitungszeit und die Rückfahrzeit zur Startposition dargestellt werden, an den Verbindungen zwischen den Maschinen soll die Transportzeit angetragen werden:

```

-----
|  1  |----->|  2  |----->...
|-----| 2s  |-----|
|  8s  |      | 10s  |
| 12s  |      | 18s  |
|  8s  |      |  6s  |
-----
...
    
```

Testen Sie mit unterschiedlichen Aufstellungsplänen und geben Sie die Graphik auch in eine Textdatei aus, deren Name interaktiv abgefragt wird.

Aufgabe 16.2 Kraftwerke

Verschiedene Kraftwerke (Wasserkraft = WKW, Kohlekraftwerk = KKW, Atomkraftwerk = AKW, Photovoltaikkraftwerk = PKW, Windkraftwerk = WKW, Pumpspeicherwerk = PSW) geben verschiedene Energiemengen zu verschiedenen Preisen ab:

	WKW	KKW	AKW	PKW	WKW	PSW
Energie (MW):	150	250	400	75	80	300
Preis (ct/kWh):	12	14	22	28	13	40

- Lesen Sie die Daten (Kraftwerkszahl, -bezeichnungen, Leistungen, Preise) aus einer Datendatei in geeignete Datenstrukturen ein.
- Ermitteln Sie den Durchschnittspreis der Energie bei gleichmäßiger prozentualer Auslastung aller Kraftwerke.
- Ermitteln Sie den günstigsten und höchsten Energiepreis für eine bestimmte Leistung. Die Leistung soll stets geringer als die Einzelleistung jedes Kraftwerks sein.
- Beachten Sie nun, dass die benötigte Leistung höher sein kann als die Leistung eines Einzelkraftwerks.

- Erweitern Sie das Modell um unterschiedliche Leistungen und Preise zu verschiedenen Tageszeiten (nachts ist es dunkel, Pumpspeicherwerke geben nur zu Hochbedarfszeiten Energie ab, zur Hauptlastzeit ist Energie teurer).
- Ermitteln Sie den günstigsten und ungünstigsten Zeitraum, um Energie zum Durchschnittspreis zu entnehmen.
- Ermitteln Sie nun für eine bestimmte Energiemenge die günstigste und ungünstigste Zeit. Welche weiteren Bedingungen müssen gestellt werden, um diese Aufgabe eindeutig zu lösen?

Aufgabe 16.3 Dampfkessel (logische Ausdrücke)

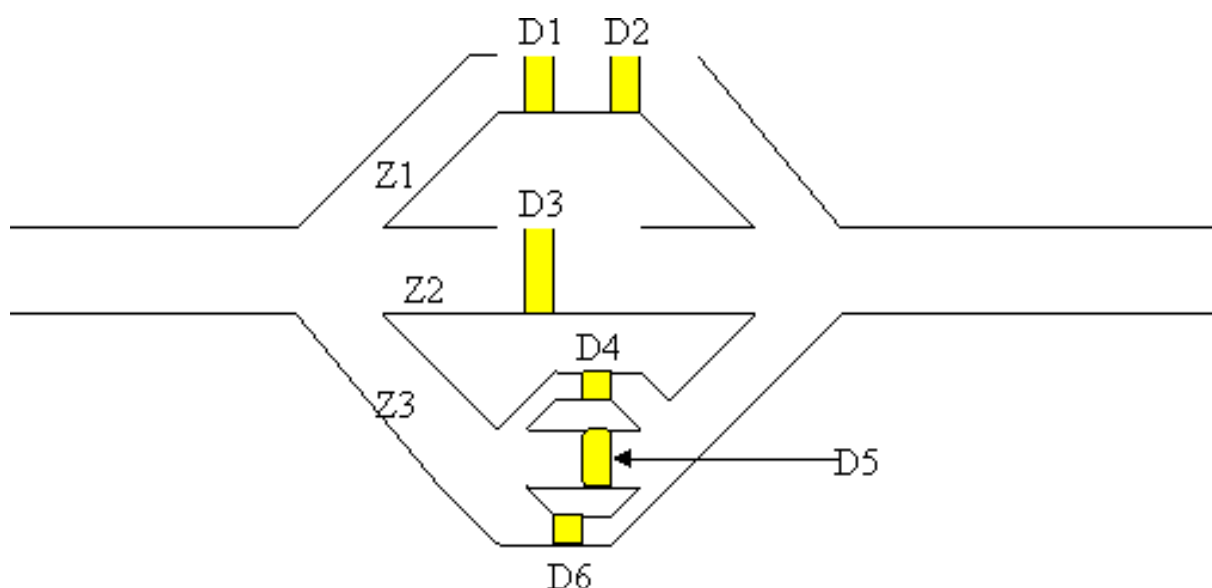
In einem Kessel sind der Druck und die Temperatur ständig zu messen, darum sind 2 Temperatur- und 3 Drucksensoren eingebaut. Oft fallen einige dieser Sensoren aus. Es ist mindestens ein Drucksensor und ein Temperatursensor nötig, um Messwerte zu nehmen.

- Schreiben Sie ein Programm, bei dem der Betreiber dieses Kessels angibt, welche Sensoren (T1,T2,D1,D2,D3) gerade funktionieren, und er soll als Ergebnis erhalten, ob alle nötigen Messwerte da sind oder nicht. Verwenden Sie genau eine *if*-Anweisung.
- Veränderte Situation. Weiterhin ist zum Temperaturmessen nur ein Sensor nötig, zum Druckmessen aber 2, da die Sensoren nicht nur ausfallen sondern auch ungenau arbeiten, darum will man sichergehen und zweimal messen. verwenden Sie wieder nur eine *if*-Anweisung.
- Nun ist es beim Temperatur messen nicht mehr egal ob einer oder 2 gehen, sondern es muss genau einer sein. Beim Druck messen müssen es genau 2 Sensoren sein, die aktiv sind.

(© Rainer Fischer)

Aufgabe 16.4 Biber (logische Ausdrücke)

Nachfolgendes Bild zeigt einen Fluss, D1 bis D6 sind Plätze an denen das Wasser flach ist und Biber ihre Dämme bauen können. Nun ist es so, dass der Fluss nicht überlaufen darf, es muss also mindestens ein Zweig (Z1 Z2 oder Z3) frei sein. Es darf maximal auch nur ein Zweig offen sein, damit der Fluss nicht oben zu wenig Wasser führt. Die Zweige Z1 und Z2 sind dann offen, wenn bei ihnen jeweils keine Dämme gebaut worden. Der Zweig 3 zählt dann als offen, wenn D5 nicht gebaut wurde, oder sowohl D4 als auch D6 offen sind. Sind D4, D5, D6 offen ist Z3 auch offen.



- Schreiben Sie ein Programm, wo der Benutzer der Reihe nach eingibt, welche Dämme gebaut worden, (Ist D1 gebaut?(j/n)).

- Eine einzige *if*-Anweisung soll klären, ob die richtige Menge Wasser fließt.

(© Rainer Fischer)

Aufgabe 16.5 *Besondere Zahlen*

Eine Zahl ist eine Fira-Zahl, wenn sie kleiner als 500 ist, durch 2, aber nicht durch 3 teilbar ist, aber auch mindestens einen von den Teilern 5 und 7 hat und entweder durch 11 oder durch 13 teilbar ist. Bestimmen Sie alle Fira-Zahlen!

(© Rainer Fischer)

Ein Fahrschein mit einer aufgedruckten sechsstelligen Nummer heißt *glücklich*, wenn die Summe der ersten drei Ziffern dieser Nummer gleich der Summe der zweiten drei Ziffern ist.

- Prüfen Sie, ob ein bestimmter Fahrschein glücklich ist.
- Ermitteln Sie die Anzahl aller glücklichen Fahrscheine.

Aufgabe 16.6 *Römische Zahlen*

Schreiben Sie Funktionen zum Umwandeln einer Zahl in Dezimaldarstellung in eine römische Zahl und umgekehrt.

Erzeugen Sie aus der Dezimaldarstellung von Zahlen bis 4000 römische Zahlen.

Rechnen Sie römische Zahlen bis 4000 in ihre Dezimaldarstellung um.

Aufgabe 16.7 *Magisches Quadrat*

Die Zahlen von 1 bis 9 sollen so auf die Felder einer 3x3-Matrix verteilt werden, dass die Summe der Zeilen, Spalten und Diagonalen gleich ist. Ermitteln Sie alle Lösungen.

Entfernen Sie alle symmetrischen Lösungen, i.e. die sich nur durch Drehen oder Spiegeln der Matrix voneinander unterscheiden.

Ermitteln Sie die Anzahl der möglichen magischen Quadrate der Größe $n \times n$, die mit den Zahlen von 1 bis n^2 gefüllt und nicht paarweise symmetrisch sind. Schätzen Sie den Aufwand ab. Was bemerken Sie?

Aufgabe 16.8 *Blumengarten*

Ein Blumenfeld ist in Quatraten aufgeteilt. Auf jedem Feld wachsen Blumen; gelbe, rote und blaue, und natürlich grünes Gras. Jedes einzelne Quadrat soll schön aussehen, sowohl allein, als auch im Vergleich zu seinen Nachbarn. Das Blumenfeld soll in C/Python mit Hilfe gleich großen Feldern (Arrays, Matrizen) abgebildet werden, für jede Farbe ein Feld. Die Feldelemente sind ganze Zahlen zwischen 0 und 10.

Das Feld für die blaue Farbe enthält also an der Position (1,1) eine Zahl zwischen 0 und 10. Diese spiegelt wider, wie viele blaue Blumen es auf dem Quadrat (1,1) des Blumenfeldes gibt usw.

- Schreiben Sie ein Methode, was die Eingabe der 4 Felder erlaubt und was darauf achtet, dass nur Zahlen von 0 bis Zehn eingegeben werden können.
- Schreiben Sie ein Methode, welches die Felder per Zufall mit Werten zwischen 0 und 10 füllt, weil das Pflanzenwachstum ja auch ein bisschen zufällig ist.
- Schreiben Sie eine Methode, die kontrolliert, dass auf jedem Blumenquadrat nur insgesamt 19 Pflanzen sind. Ist das bei einem Feld nicht der Fall, dann geben Sie als Bildschirmausgabe an, wo der Gärtner den Rasenmäher ansetzen muss. Die Methode soll auch gleich noch die nötigen Pflanzen entfernen, dabei werden immer zuerst die gelben Blumen weggeschnitten (es ist ja meist nur Löwenzahn), dann die blauen, dann die roten, solange bis weniger als 19 Blumen auf einem Quadrat sind.

- Schreiben Sie eine Methode, die prüft, ob ein Feld schön ist. Dazu muss es mindestens halb so viel grünes Gras haben, wie bunte Blumen. Ansonsten muss Gras gesät werden (jetzt dürfen auch wieder mehr als 19 Pflanzen auf einem Feld sein, es ist ja nur Gras!)
- Schreiben Sie eine Methode, die prüft, dass ein Feld zu seinen Nachbarn passt. Nachbarn sind alle 8 angrenzenden Felder. Ein Feld passt zu seinen Nachbarn, wenn es von einer Blumenfarbe (nicht Gras), nur maximal 5 mehr hat, als der Nachbar mit der geringsten Anzahl dieser Blumen. Ansonsten muss die Anzahl an Blumen reduziert werden (Rasenmäher!)
- Benutzen Sie statt der vier Felder ein Feld von Objekten, wobei jedes Objekt die Bepflanzung (Anzahl der Blumen jeder Farbe) repräsentiert.
- Benutzen Sie ein dreidimensionales Feld statt der vier Einzelfelder.

(© Rainer Fischer)

Aufgabe 16.9 Folgen

Programmieren Sie *Langton's Ant*. Eine Ameise befindet sich auf einer weißen karierten Fläche und bewegt sich zunächst nach Norden. Jedes Feld, dass sie verlässt wechselt seine Farbe von weiß nach schwarz bzw. schwarz nach weiß. Gelangt die Ameise auf ein weißes Feld, geht sie im nächsten Schritt nach links, gelangt sie auf ein schwarzes Feld, geht sie im nächsten Schritt nach rechts.

Aufgabe 16.10 Sortieren und Suchen

Ein Handwerker bekommt an einem festen Starttag eine Menge von Aufträgen, wobei jeder Auftrag eine feste und ganze Anzahl von Tagen umfasst. Dabei muss der Handwerker jeden Auftrag hintereinander komplett erledigen. Allerdings muss er pro Tag Verzögerung gegenüber dem Starttag eine bestimmte, vom Auftrag abhängige Strafe zahlen (ebenfalls ganzzahlig). Schreiben Sie ein Programm, dass die Reihenfolge der Aufträge so bestimmt, dass die Strafe minimiert wird.

Beispiel: 4 Aufträge.

```
Auftrag 1: 3 Tage, Strafe: 4
Auftrag 2: 1 Tag, Strafe: 1000
Auftrag 3: 2 Tage, Strafe: 2
Auftrag 4: 5 Tage, Strafe: 5
```

Eine beste Reihenfolge (es gibt 2) ist:

```
Reihenfolge:  2 1 3 4
Beginn (Tag): 0 1 4 6

Strafe:
Auftrag 2: 0T * 1000 =  0
Auftrag 1: 1T *  4   =  4
Auftrag 3: 4T *  2   =  8
Auftrag 4: 6T *  5   = 30
-----
Summe:                42
```

Hinweis: Untersuchen Sie, wie sich die Strafe ändert, wenn Sie in einer vorgegebenen Reihenfolge zwei benachbarte Aufträge vertauschen.

Aufgabe 16.11 Zahlenraten

Schreiben Sie ein Programm, dass eine Zahl zwischen 1 und N mit möglichst wenigen Versuchen rät. Das Programm rät eine Zahl und erhält als Antwort:

- 0: erster Versuch oder Abstand zur gesuchten Zahl ist genauso groß wie beim vorangegangenen Versuch,

- 1: neue geratene Zahl liegt dichter an der gesuchten Zahl als beim vorigen Versuch,
- -1: neue geratene Zahl ist weiter entfernt von der gesuchten Zahl als beim vorigen Versuch.

Wird eine geratene Zahl wiederholt, muss das die gesuchte Zahl sein, das Programm wird beendet.

Schreiben Sie auch die Auswertung als Programm!

Beispiel: Zahl zwischen 1 und 10 raten, gesucht sei 5.

```
7 Antwort: 0 (Abstand 2)
9 Antwort: -1 (Abstand 4 - schlechter)
4 Antwort: 1 (Abstand 1 - besser)
6 Antwort: 0 (Abstand 1 - gleich)
5 Antwort: 1 (Abstand 0 - besser)
5 Ende (korrekt)
```

Hinweis: Die im Beispiel benutzte Strategie ist nicht optimal.

Aufgabe 16.12 *Strings und Modulorechnung*

Verschlüsseln Sie einen Text, der nur aus Buchstaben besteht, nach der Cäsar-Chiffre: Verschieben Sie jeden Buchstaben um k Positionen (für $k=3$ wird aus A ein D, aus B ein E etc.). Wird das Z überschritten, wird wieder bei A begonnen (aus X wird A, aus Y wird B, aus Z wird C).

Sowohl der Text als auch k sollen eingegeben werden können.

Verschlüsseln Sie einen Text, der nur aus Buchstaben besteht, nach der Vigenère-Chiffre: Wir betrachten die Buchstaben eines Schlüssels, z.B. "KEY", als Zahlenwerte mit A=0, B=1. Nun verschieben wir den ersten Buchstaben der Nachricht um den ersten Zahlenwert des Schlüssels, den zweiten um den zweiten Wert etc. Ist das letzte Zeichen des Schlüssels abgearbeitet, beginnen wir wieder von vorn.

16.2 Adventskalender der TU Chemnitz

Der [Adventskalender der TU](#) enthielt jedes Jahr ein Logikrätsel, das sich meist per Computer lösen lässt.

Meist ist ein rekursives Vorgehen mittels Tiefensuche möglich, wobei teilweise geeignete Strategien zur Verkleinerung des Suchbaums eingesetzt werden müssen (Pruning: Abschneiden aussichtsloser Zweige).

- [Schiebefax](#)
- [Rentier Rudolph](#)
- [Kaputtes Reiterlein](#) (schwierig)
- [Verstecktes Weihnachtsbild](#) (Hinweis: unten beginnen)
- [Sudoku](#)
- [Knecht Ruprecht sucht den Weg](#)
- [Bertolt sucht den Ausgang](#)
- [Geschenkröhrpost](#)
- [Das kippt](#)
- [Hausversorgung](#)
- [Das rutscht](#)
- [Labyrinth](#) (auch mit Breitensuche lösbar, Anzahl Spielsituationen bestimmen)
- [Drehtüren](#)
- [Planverschiebung](#)
- [Staubsaugen](#)

- Linie 3
- [Home Automation](#) (sehr großer Suchraum, mit einfachem Backtracking nicht lösbar)
- [Frachtflüge](#) (erste Lösungen sind schnell zu finden, komplettes Durchmustern des Suchraums aufwendig)

16.3 Aufgabensammlungen

- [Programming Challenges](#) – die Aufgaben können eingesandt werden, es erfolgt eine automatische Korrektur.
- [Project Euler](#)
- [hacker.org](#) (Anmeldung erforderlich, selbst noch nicht getestet)

Begriffe und Abkürzungen

ASCII *American Standard Code for Information Interchange*

Internationaler Zeichensatz, der mit 7 Bit codiert werden. Er enthält die lateinischen Buchstaben, Ziffern und wichtigsten Satzzeichen, aber keine akzentuierten Buchstaben der europäischen Sprachen.

AuP *Algorithmen und Programmierung*

Lehrveranstaltung des 1. Semesters.

CSV *Comma Separated Values*

Einfache Darstellungsform von Tabellen in Textdateien. Die Felder werden durch Komma (bzw. in deutschen Anwendungen durch Semikolon) getrennt.

GCC *GNU* Compiler collection

GNU *GNU is not Unix*

Rekursives Acronym

GPU *Graphic Processing Unit*

Die Prozessoren der Graphikkarte, die für Berechnungen mit speziellen Bibliotheken zweckentfremdet genutzt werden können. Diese sind bei bestimmten Aufgaben z. B. des Machine Learnings deutlich schneller als der eigentliche Prozessor des PCs.

IBAN *International Bank Account Number*

EU-einheitliche Kontonummer.

IDE *Integrated Development Environment*

Integrierte Entwicklungsumgebung: komfortabler Editor mit Anbindung der Entwicklungswerkzeuge und Dokumentation.

JSON *JavaScript Objects over Network*

Textbasiertes Datenformat zum Speichern strukturierter Daten. Der Name ergibt sich daraus, dass JSON-Dateien ausführbaren JavaScript-Code darstellen.

ISBN *International Standard Book Number*

JVM *Java Virtual Machine*

Abstraktion des konkreten Prozessors und Betriebssystems durch einen genau definierten virtuellen Computer, der durch einen Interpreter in das reale System übersetzt wird.

LLVM *Low Level Virtual Machine*

OOP *Object Oriented Programming*

PHP *PHP: Hypertext Preprocessor*

Beliebte Programmiersprache des [WWW](#)

WASM *WebAssembly*

Bytecode für die Ausführung im Webbrowser.

WWW *World Wide Web*

Über Internet abrufbares System von Hypertext-Dokumenten, das »Internet« im üblichen Sinne.

Literaturverzeichnis

- [Ernesti20] Ernesti, Kaiser: Python 3, Das umfassende Handbuch. 6. Auflage, Rheinwerk 2020.
Freie Online-Version [Python-Buch](#)
- [KR90] Kernighan, Ritchie: Programmieren in C. 2. Auflage, Hanser 1990
Das Standardwerk, gut verständlich, als Lehrbuch geeignet.
- [Kofler18] Kofler: Python, Der Grundkurs. Rheinwerk 2018.
Umfangreiche und recht preiswerte Einführung.
- [Kopeck20] Kopeck: Algorithmen in Python. Rheinwerk 2020.
Vorstellung wichtiger Algorithmenklassen mit Beispielimplementierung in Python.
- [SPR.C1] RRZN: Die Programmiersprache C. RRZN 1999.
Preiswert im URZ erhältlich (3,50 €)
- [Sed14] Sedgewick, Wayne: Algorithmen. 4. Auflage. Pearson 2014.
Oft zitiertes Buch, das eine große Zahl wichtiger Algorithmen vorstellt. Neben dieser generischen Ausgabe auch angepasste Versionen für verschiedene Programmiersprachen erhältlich.
- [WoKr20] Wolf, Krooß: Grundkurs C. 3. Auflage, Rheinwerk 2020.
Umfangreiche und recht preiswerte Einführung.
- [Wolf20] Wolf, Krooß: C von A bis Z. 3. Auflage, Rheinwerk 2020.
Freie Online-Version [C-Buch](#)

A

Abweisende Schleife, 45, 47
Adjazenzmatrix, 126
Arbeitsverzeichnis, 3
ASCII, 201
assert(), 29
Assertion, 29
AuP, 201
Ausdruck, 11
Ausgabe
 Python, 144

B

Bezeichner, 19
Block, 9
bool, 35
break, 49
Buffer Overflow, 100

C

Cast, 55
char, 97
char-Feld, 98
Compiler, 2, 5
 Aufruf, 10
continue, 49
C-Programm
 Übersetzung, 10
CSV, 201

D

Data Frame, 131
Dateiumlenkung, 14
Datenkapselung, 83
Datentyp
 bool, 35
 char, 97
 double, 53
 int, 11
 struct, 83
Definitionsbereich, 23
double, 53
do-while, 48

E

Editor, 5
Eingabe, 14
Einlesen von Strings, 100
enum, 17

F

Feld
 Python, 157
Felder
 Sortieren, 109
Fester Wert, 20
for, 47
Funktion, 23
 Python, 142

G

GCC, 201
Gewicht, 125
GNU, 201
GPU, 201
Graph, 123, 125
 bipartit, 125, 131
 Modellierung, 126

H

Heron-Verfahren, 61

I

IBAN, 201
IDE, 201
Identifizier, 19
Import, 153
Insertion Sort, 113
int, 11
Interpreter, 2
ISBN, 201

J

JSON, 201
JVM, 201

K

Kante, 125

Kapselung, 180

Klasse

Definition, 179

Getter, 182

Methode, 180

Setter, 182

Klassen, 177

Knoten, 125

Kommandozeile, 4

Kommandozeilenparameter

Python, 154

Kommentar, 12

Konsole, 4

Konstante, 17

L

Liste, 157

Durchlaufen, 159

Elementzugriff, 159

Erzeugen, 159

Funktionsparameter, 161

Rückgabewert, 161

Verändern, 159

Literal, 20

LLVM, 202

M

main (*C-Funktion*), 10

Maschinensprache, 2

Mathematikbibliothek, 59

Merge Sort, 114

Methode, 180

Module, 153

N

Newton-Verfahren, 61

Nichtabweisende Schleife, 48

Nullstellenberechnung, 61

O

Objektorientierte Programmierung, 177

OOP, 177, 202

P

PHP, 202

printf (*C-Funktion*), 10

Programm, 2

Programmiersprache, 2

Python

Einführung, 139

Funktionsdefinition, 142

Programmstruktur, 141

Python-Klasse, 179

Q

Quadratwurzel, 61

Quelltext, 2

Quicksort, 114

R

Rekursion, 30

S

Schleife, 45

abweisend, 45, 47

do-while, 48

for, 47

nichtabweisend, 48

Steuerung, 49

while, 45

Schleifensteuerung, 49

Selection Sort, 112

Shell, 4

Sortieren, 109

Stringfeld, 118

Struct, 120

Sortiervverfahren, 109

Insertion Sort, 113

Merge Sort, 114

Quicksort, 114

schnelle, 114

Selection Sort, 112

Standardbibliothek

Python, 153

String, 98

Einlesen, 100

Einlesen (*Python*), 171

Großbuchstaben (*Python*), 174

Länge (*Python*), 171

Listen (*Python*), 173

Literal (*Python*), 171

Typumwandlung (*Python*), 174

Vergleich (*Python*), 171

Verketteten (*Python*), 171

Zeichen (*Python*), 171

Zerlegen, 105

Zerlegen (*Python*), 174

Zusammensetzen (*Python*), 174

Stringfeld, 101

Stringlisten, 173

Stringliteral, 98

Strings, 169

struct, 83

Ein- und Ausgabe, 88

Funktion, 86

Funktionsparameter, 86

Komponente, 83

Rückgabetyt, 86

Stringkomponente, 105

T

Taylorreihe, 61

Test, 25

Tokenizer, 105

Typkonvertierung, 55

Python, 154

U

Übersetzung

 Compilerschalter, 10

 C-Programm, 10

Unicode, 169

V

Variable, 11

Verkürzte Zuweisung, 47

Verzeichnis, 3

Verzweigungen, 25

W

WASM, 202

Wertebereich, 23

while, 45

WWW, 202

Z

Zeichencode

 Python, 173

Zeichenfeld, 98

Zeichenliteral, 97

Zeichentyp, 97

Zusicherung, 29

Zuweisung, 11

 verkürzt, 47