GRUNDLAGEN DER MODERNEN INFORMATIONSTECHNIK 2

SOSE 2018

Lehrstuhl für Automatisierung und Informationssysteme

Prof. Dr.-Ing. B. Vogel-Heuser

Inhalt

Seite

Bei organisatorischen Fragen zur Lehrveranstaltung wenden Sie sich bitte an die E-Mail it-orga@ais.mw.tum.de oder direkt an die Lehrstuhl-Mitarbeiter.

Bei inhaltlichen Fragen zur C-Programmierung oder Anlagen-Programmierung wenden Sie sich bitte an die Tutoren in der Sprechstunde.

Quelloffenes Lehrbuch

Programmieren in C

Herausgegeben von B. Vogel-Heuser

19. Februar 2018

Lehrstuhl für Automatisierung und Informationssysteme
Technische Universität München

B. Vogel-Heuser (Hrsg.). Programmieren in C. Quelloffenes Lehrbuch. Technische Universität München. Garching bei München, 2018.

Dieses Buch ist ein Werk des Projekts *eTeachingKit*, welches offene Materialien und offene Software für Programmierkurse entwickelt. Machen Sie mit und unterstützen Sie unsere Idee von guten universitären Kursen für Jedermann. Besuchen Sie uns unter `http://eteachingkit.sourceforge.net`!

Autoren: K. Centmayer, F. Obermeyer, B. Tratz, S. Lauf, W. Bamberger, Timo Frank, Sebastian Rehberger, Niklas Seubert
Auflage: 1.1
Kontakt: `it-orga@ais.mw.tum.de`
Druck: Fachschaft Maschinenbau e.V.

Inhaltsverzeichnis

1 Arbeitsumgebung

1.1 CodeBlocks

1.1.1 Download und Installation

1. Die Installationsdatei kann unter *http://www.codeblocks.org/downloads/binaries* heruntergeladen werden. **Wichtig!** Sie müssen hierbei immer die größere Datei herunterladen, da diese den Compiler beinhaltet.
2. Starten Sie nach dem Download mit einem Doppelklick auf die Datei den Setup.
3. Folgen Sie nun dem Installationsassistenten durch die Installation.
4. Im Auswahlmenü für die Komponenten, können Sie die Auswahl auf Standard belassen.
5. Im nächsten Fenster haben Sie die Möglichkeit Ihren gewünschten Installationspfad auszuwählen.
6. Mit einem Klick auf *Install* starten Sie die Installation.
7. Die Installation ist nun abgeschlossen und CodeBlocks kann gestartet werden.

1.1.2 Das erste Programm

1. Starten Sie Codeblocks mit einem Doppelklick auf die *codeblocks.exe*. Diese finden Sie in Ihrem gewählten Installationsordner.
2. Es erscheint nun ein Fenster, in welchem verschiedene Compiler aufgelistet sind. Setzen Sie den *GNU GCC Compiler* als Standard, indem Sie zunächst auf den Namen und dann auf *Set as default* klicken. Klicken Sie anschließend auf *OK*.
3. Im nächsten Fenster erscheint nun ein *Tip of the Day*, diesen Tipp können Sie schließen. Es erscheint nun die Frage, ob Sie Codeblocks als Standard Programm für C/C++ Dateien verwenden möchten. Falls Sie das möchten, was zu empfehlen ist, wählen Sie *Yes, associate Code::Blocks with C/C++ file types* und klicken Sie auf *OK*. Die *Scripting console* können Sie ebenfalls schließen.
4. Sie müssen nun noch einige Einstellungen am Compiler vornehmen. Diese Einstellungen finden Sie im Moodle-Kurs unter *CodeBlocks-Tutorial*.

5. Um ein neues Programm zu erstellen, wählen Sie links oben *File -> New -> File...* aus. Im nächsten Fenster wählen Sie unter dem Reiter *Files* die *C/C++ Source* aus und klicken auf *Go*.

6. Das nächste Fenster können Sie mit einem Klick auf *Next >* überspringen.

7. Wählen Sie nun *C* aus und klicken sie wieder auf *Next >*.

8. Durch einen Klick auf *...* können Sie einen Pfad wählen, in welchem CodeBlocks Ihr File speichern wird. Geben Sie nach der Auswahl des Pfades noch einen Dateinamen ein und drücken Sie auf *Speichern*. CodeBlocks ergänzt die Endung .c automatisch. Klicken Sie jetzt noch auf *Finish* und Ihr erstes File wird erstellt.

9. Nun können Sie Ihren ersten Code eintippen. Tippen Sie vielleicht als kleinen Test, das folgende Programm ab:

```
#include <stdio.h>

int main ()
{
    printf ("Hallo Welt!");
    return 0;
}
```

   Klicken Sie nun auf *Build and run*, hierdurch wird Ihr Programm kompiliert und gestartet. Zum kompilieren können Sie auch *Strg+F9* und anschließend *Strg+F10* drücken. Es öffnet sich nun die Konsole und Sie sollten *Hallo Welt!* lesen können.

10. Am unteren Bildschirmrand, im *Build log*, finden Sie eventuelle Fehler und Warnungen.

11. Sie haben nun Ihr erstes Programm mit CodeBlocks erstellt. CodeBlocks bietet noch viele weitere Funktionen, die Sie nun selbst durch ausprobieren entdecken können.

1.2 Moodle

Die Testate sowie das Anlagenpraktikum werden in Moodle abgenommen. Zur inhaltlichen Vorbereitung sowie der Gewöhnung an die Benutzeroberfläche werden Ihnen Heimarbeitsblätter in Moodle zur Verfügung gestellt.

2 Einführung in C

Wie Sie bereits aus der Vorlesung wissen, lernen Sie im Fach IT-2, die Programmiersprache C kennen.

Wie sieht nun ein einfaches Programm in C aus?

2.1 Ein erstes, einfaches Beispiel

Betrachten wir einmal das Programm *happy.c*. Dabei werden einige leichte Fragen gestellt.

```
#include <stdio.h>

int  main()
{
   printf ("Hurra, hurra!");
   printf ("\n");
   printf ("Asterix ist  da!\n");
   return 0;
}
```

Listing 2.1: happy

Jedes Programm besteht aus einem Hauptprogramm `main` und eventuell weiteren Modulen. Das Hauptprogramm heißt in *C* immer `main`. Dieser Programmtext ist in einer Datei abgespeichert, sie hat den Namen *happy.c*.

Wie heißt unser Hauptmodul? Unsere Datei trägt den Namen *happy.c*. Das Hauptprogramm selbst trägt immer den Namen `main`. Es ist zu bemerken, dass in *C* zwischen Groß- und Kleinschreibung unterschieden wird. Achtung: dies gilt (zumindest beim Übersetzen unter *UNIX*) auch für die Dateinamen!

Wie wir sehen, wird mit `#include` etwas in unser Modul eingefügt, nämlich die erforderliche Information für die Ein/Ausgabe (I/O). Zur Ein/Ausgabe später mehr (vor allem in Kapitel 4), aber was wird hier eingefügt?

Die Zeile `#include <stdio.h>` fügt die Datei *stdio.h* in das Programm ein, in der u.a. die benötigte Information zur Verwendung der Ein-/Ausgabe-Anweisung `printf` enthal-

ten ist. Dieses Einfügen wird auf rein textueller Ebene gemacht – es wird also wirklich (vor dem Übersetzen) diese Datei in den Quelltext einkopiert! Dieser Schritt wird übrigens vom Präprozessor übernommen (erkennbar an dem #-Zeichen: diese Anweisungen wenden sich an den Präprozessor), für den Programmierer ist das aber eigentlich egal.

`printf("Hurra")` gibt den Text `Hurra` am Bildschirm aus, `printf("\n")` bedeutet, dass zu einer neuen Zeile gesprungen wird (es wird ein Zeilenumbruch eingefügt).

Hier das Programm *happy.c* mit Kommentaren:

```
#include <stdio.h>              /* Einfügungen   */

int main()                      /* Programm-Kopf */
{
  printf ("Hurra, hurra!");     /*       -       */
  printf ("\n");                /* Programm-Block */
  printf ("Asterix ist da!\n"); /*       -       */
  return 0;                     /*       -       */
}/* End Happy */                /* Kommentar     */
```

Listing 2.2: happy mit Kommentaren

Was bedeutet aber die Zeile /* End Happy */? `/* End Happy */` bedeutet für das eigentliche Programm gar nichts, es wird vom Rechner überlesen. Es ist nur ein Kommentar für den menschlichen Leser des Programmes. Also nochmal zusammenfassend: Text, der in `/* bel. Kommentartext */` eingeschlossen ist, wird vom Compiler überlesen.

Ein *C*-Programm besteht also aus mehreren Teilen:

- Einfügungen
- dem Programm-Kopf mit dem Schlüsselwort main
- dem Programm-Block
- Kommentaren

Der eigentliche Programm-Block, in dem die abzuarbeitenden Anweisungen stehen, wird in geschweiften Klammern eingeschlossen:

```
{ Programm-Block }.
```

Die geschweiften Klammern `{ }` entsprechen in *C* also einem `begin ... end`, das aus anderen Programmiersprachen eventuell bekannt ist.

Wir sehen, dass der Programmaufbau bestimmten Regeln gehorcht, den sogenannten Syntaxregeln. Die Syntaxregeln werden hier in der sogenannten *Erweiterten Backus-Naur-Form* (EBNF) dargestellt.

Bemerkung: Mit dem Skript erhalten Sie einige Quellcode Beispiele. Diese Beispiele finden Sie in Moodle unter den ergänzenden Dokumenten zum Skript als Text-Datei. Den Namen der entsprechenden Datei finden Sie immer in der Beschreibung unter dem Quellcode im Skript. In Kapitel 2 steht das oben als *happy* bezeichnete Programm unter dem Namen *happy.txt* zur Verfügung.

2.2 Programmentwicklung

2.2.1 Schritte

Bevor wir nun mit den Details der Programmiersprache beginnen, sollten noch einige allgemeine Hinweise zum Programmieren gegeben werden.

Welche Schritte sind bei der Programmierung überhaupt durchzuführen? Nehmen wir wieder das Beispiel *happy.c* und betrachten Sie dazu Abbildung 2.1 auf der nächsten Seite:

1. Erstellung des Quell-Programmes, d.h. der Datei *happy.c*

   Dieser Schritt versteht sich eigentlich von selbst: man muss den Quelltext zunächst mal erstellen, und meist auch noch mehrfach überarbeiten.

2. Übersetzen (compilieren) des Programmes (erzeugt die Datei *happy.o*)

   In diesem Schritt wird der Quelltext in Maschinensprache übersetzt. Dabei „entdeckt“ der Compiler schon ein Vielzahl möglicher Fehler. Falls der Quelltext nicht mit den Regeln der Sprache vereinbar ist, „weiß“ der Compiler nicht, was mit dem Code anzufangen ist und er bricht das Übersetzen mit einer Fehlermeldung ab. Dann muss das Programm entsprechend korrigiert werden (Schritt 1). Bei verdächtig aussehenden Konstruktionen wird häufig eine Warnung ausgegeben, aber das Programm dennoch übersetzt. Diese Warnungen sind oft ein guter Hinweis auf wirkliche Fehler – also besser nochmal zu Schritt 1 gehen. Siehe auch Anhang: Fehlermeldungen und Warnungen des Compilers.

3. Linken des Programmes (erzeugt die ausführbare Datei *happy*)

   Hier werden alle Bestandteile des Programms zusammengebunden. Wenn alle nötigen Teile gefunden wurden, entsteht ein ausführbares Programm. Andernfalls müssen evtl. noch die Linker-Einstellungen angepasst und neu gelinkt werden (Schritt 3), oder es sind Fehler in der Quelldatei zu korrigieren (Schritt 1).

4. Starten des Programmes

   Beim Ablauf des Programmes können ebenfalls Fehler auftreten, dann muss erneut (1, . . . , 4) korrigiert, compiliert und gelinkt werden.

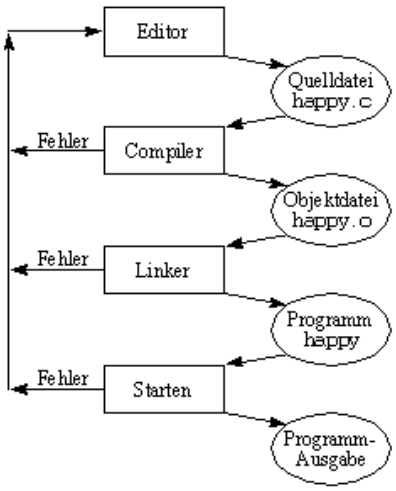


Abbildung 2.1: Schritte bei der Programmerzeugung

Ein Programm, das nun mehrfach verwendet werden soll, um mit unterschiedlichen Daten zu arbeiten, muss unbedingt ausgetestet werden; das heißt, die berechneten Ergebnisse müssen sorgfältig überprüft werden. Man muss also die Berechnung anhand von einigen Beispielen nachprüfen, damit man sicher sein kann, dass weitere Berechnungen richtig sind.

Vergessen Sie auch nicht, dass in C zwischen Groß- und Kleinschreibung unterschieden wird!

2.2.2 Programmierstil

Auch wenn Ihnen vermutlich einige nachfolgend aufgeführten Punkte noch nichts sagen werden, werden wir dennoch gleich zu Beginn auf einige stilistische Fragen bei der Programmentwicklung eingehen. Es macht einfach mehr Sinn, diese Punkte von Anfang an zu beachten, als sich dann später erst umstellen zu müssen.

Achten Sie bereits bei der Entwicklung Ihrer Programme auf guten Programmierstil. Folgende Punkte sind dabei besonders wichtig:

- Fügen Sie Kommentare ein.
- Schreiben Sie den Quellcode strukturiert, d.h. eine Anweisung pro Zeile, Einrücken der Programmblöcke in Verzweigungen und Schleifen.

- Verwenden Sie aussagekräftige Namen.
- Keine zu komplexen Anweisungen, lieber mehrere Anweisungen. Der Rechner benötigt dies natürlich nicht, der Compiler erwartet nur eine korrekte Syntax; aber das Programm wird dadurch lesbarer und die Fehlersuche wird erleichtert.
- Achten Sie auf eine übersichtliche Ablaufsteuerung. Vermeiden Sie insbesondere `goto`-Anweisungen.

Diese Punkte können im Grunde gar nicht genug hervorgehoben werden. Es ist immer wieder überraschend, mit wie schlecht geschriebenem Code (den natürlich immer andere geschrieben haben . . .) man sich in der Praxis ärgern muss.

Kommentare sollen nicht die Programmiersprache selbst erklären, sondern die Idee erläutern, die hinter einem Code-Abschnitt steckt.Vor allem die Frage warum ist das jetzt so oder so „programmiert“ sollte erläutert werden. Am besten man denkt daran, wie man den Code jemandem erklären würde, der ihn zum ersten mal liest, aber sehr wohl mit C vertraut ist. Dies hilft dann vor allem dem Programmierer selbst – viele Fehler sind schon allein durch den Versuch entdeckt werden, den fraglichen Codeabschnitt zu erklären. Scheuen Sie sich auch nicht, eigene Merkhilfen oder Ähnliches als Kommentar zu schreiben!

Auch die (für C an sich belanglose) Formatierung des Quelltexts hat entscheidenden Einfluss auf die Verständlichkeit des Codes. Überlange Zeilen, die vollgestopft sind mit schwer zu verstehenden Ausdrücken, beschwören Fehler geradezu herauf. Der Programmierer tut sich damit selbst keinen Gefallen. In größeren Projekten wird häufig sogar exakt festgelegt, wie die Codeblöcke einzurücken sind, an welcher Stelle die geschweiften Klammern stehen, ob Leerzeichen vor und nach einer Klammer oder einem Gleichheitszeichen stehen usw. Bei den Heimarbeiten und Testaten arbeiten Sie (mehr oder weniger) allein an Ihren Programmen, also können Sie diese Regeln auch selbst bestimmen – aber Sie sollten darauf achten, dass das Ergebnis einheitlich und übersichtlich ist.

Die Vergabe von Namen für Funktionen und Variablen kann für die Verständlichkeit von Programmen noch wichtiger sein als Kommentare. Wenn Code durch aussagekräftige Namen schon selbsterklärend ist, kann man auch mal einen Kommentar einsparen. Aber die Namen sollten auch nicht zu lange werden, es ist ein Kompromiss zu schließen zwischen Verständlichkeit und Schreibarbeit.

Sehr komplizierte und ineinander verschachtelte Ausdrücke lassen sich häufig durch Verwendung von „Zwischenrechnungen“ wesentlich vereinfachen. Sparen Sie auch nicht mit Klammern! Man kann zwar auch hier übertreiben, aber lieber mal eine Klammer mehr, und der Ausdruck wird wesentlich leserlicher. Machen Sie sich keine Sorgen um die Laufzeit (Geschwindigkeit) des Programms, die ist im Rahmen der Heimarbeiten und Testate nicht wichtig. Moderne Compiler verfügen über ausgeklügelte Optimierungsstufen, die ein Optimieren einzelner Ausdrücke sowieso besser beherrschen.

Auch die Ablaufsteuerung des Programms kann übersichtlich oder weniger übersichtlich sein. Viele „trickreiche“ Programmabschnitte mit Sprüngen an allen möglichen Stellen, tief verschachtelten Schleifen und hunderte von Zeilen umfassenden Funktionen sind später schlicht nicht mehr zu gebrauchen, da man keine Fehler findet und keine Änderungen vornehmen kann.

Zusammenfassend kann man die einfache Faustregel aufstellen, dass Code übersichtlich, verständlich und auch für andere lesbar sein sollte. Dadurch werden bereits viele Fehler vermieden, die dann später nicht mehr gesucht werden müssen. Mehr dazu in Kapitel 7.

2.2.3 Fehlersuche

A propos Fehlersuche („Debugging“): Erfahrene Programmierer benutzen hierfür meist Debugger, die eine genaue Analyse der Abläufe in einem Programm ermöglichen (siehe auch: Kapitel A.1.4). Fügen Sie einfach an geeigneten Stellen Ausgaben ein, die z.B. den Wert einer Zwischenvariable oder eines Schleifenzählers ausgeben. Wenn das Programm korrekt funktioniert, können Sie diese Anweisungen ja einfach wieder entfernen.

Weitere Informationen zu diesem Thema finden Sie in Anhang A.

Vor der Fehlersuche steht aber natürlich das Programmieren selbst. Woraus besteht nun genau ein C-Programm?

2.3 Grundlegende syntaktische Begriffe

2.3.1 Zeichen und Symbole

Ein Programm besteht aus einer Folge von Zeichen, die nach bestimmten, für die jeweilige Programmiersprache charakteristischen Vorschriften verknüpft werden. Die Verknüpfungsregeln für ein Programm bezeichnet man als die *Syntax* der Programmiersprache. Die Syntax kann man durch bestimmte Begriffe wie Ausdruck, Prozedur, Anweisung usw. beschreiben. Jedes Programm besteht aus *Zeichen*, die der Benutzer mit Hilfe eines Editier-Programms eingeben muss. Jedes dieser Zeichen wird im Inneren des Rechners durch ein bestimmtes Bitmuster kodiert. Diese Zeichen werden nach bestimmten, für die Programmiersprache typischen Regeln zu einem Programm verknüpft. Sehen wir uns nun diese Zeichen im einzelnen an. Der Zeichenvorrat von C umfasst Buchstaben, Ziffern und bestimmte Sonderzeichen. (s. Tabelle: *Zeichen und Symbole*)

Zu den Sonderzeichen zählt auch das Leerzeichen. Alle die vorher erwähnten Zeichen dürfen in einem C-Programm vorkommen. Einzelne oder mehrere (in sinnvoller Weise zusammengefasste) Zeichen bilden Symbole.

Ein *Symbol* hat im Programm eine bestimmte Bedeutung, es kann einen Bezeichner oder Operator darstellen. Ein *Operator* kann aus einem (`+ - * /` usw.) oder mehreren Zeichen bestehen. Den Operator „ungleich“ z.B. schreibt man als `!=`.

Klein- und Großbuchstaben	a, ..., z, A, ..., Z
Ziffern	0, 1, ..., 9
Leerraum	Leerzeichen, Tabulator und Zeilenende
Sonderzeichen	+ - * / = & # % ^ ~ < > ( ) [ ] { } ! \ " ' | . , ; ? : _

Tabelle 2.1: Zeichen und Symbole

Wie sieht dann das Symbol für „größer gleich“ aus? Es lautet >=. Für „kleiner gleich“ entsprechend <=.

2.3.2 Reservierte Wörter

Hier sehen Sie die Liste der reservierten Wörter (Wortsymbole), die *nicht* zur Bezeichnung anderer Größen verwendet werden dürfen. Diese Symbole werden auch als *Schlüsselwörter* (*keywords*) bezeichnet.

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

Tabelle 2.2: Keywords

2.3.3 Namen

Wir haben bisher die zur Beschreibung einer Programmiersprache wichtigen Begriffe Zeichen und Symbol kennengelernt. Ein weiterer Begriff ist der des *Namens* (oder *Bezeichners*). Namen (engl. *identifier*) werden zur Bezeichnung gewisser Größen, z.B. Variablen, benötigt.

Ein Name besteht aus einer Folge von Buchstaben und Ziffern, wobei das erste Zeichen ein Buchstabe sein muss.

Darf man also die Zeichenfolge „x-Wert“ als Name für eine Variable benutzen? In einem Namen dürfen natürlich nur Buchstaben und Ziffern, sowie der Unterstrich (_), aber z.B. kein Minus-Zeichen vorkommen, wobei das erste Zeichen des Namens ein Buchstabe sein muss. Nicht erlaubt sind auch Anführungszeichen, Klammern oder Leerzeichen als Namensbestandteile.

Den Begriff Name kann man mittels EBNF als

```
Name ::= Buchstabe {Buchstabe | Ziffer}
```

darstellen.

Sind also die folgenden Zeichenfolgen sämtlich korrekte Namen im Sinne von C?

```
hoehe   N31   DM101   2n   x4   Laenge   3X   A5B
```

Fast alle. Nur `2n` und `3X` beginnen mit einer Ziffer und sind daher keine gültigen Namen.

2.3.4 Zeichen und Zeichenketten

Betrachten wir als nächstes gleich die Begriffe *Zeichenkette* (*string*) und *Zeichen* (*character*). Ein einzelnes Zeichen, z.B. der Buchstabe A, wird zwischen Apostrophe ' gesetzt. Nun handelt es sich nicht mehr um den Namen (identifier) `A`, sondern um ein Zeichen.

Sonderzeichen, nicht druckbare Zeichen, können als *escape sequence* mit \ (backslash) eingegeben werden.

Beispiele sind:

`\n`	newline (neue Zeile)
`\f`	formfeed (neue Seite)
`\t`	horizontal tab (Tabulator)
aber auch:	
`\'`	single quote (Apostroph)
`\"`	double quote (Anführungszeichen)
`\\`	backslash (Rückstrich)
`\ooo`	octal value (beliebiges ASCII-Zeichen in oktaler Kodierung)

Tabelle 2.3: Beispiele für Escape-Sequenzen

Ein String ist eine Folge von beliebigen Zeichen, die zwischen Anführungszeichen " (Doppelhochkomma) stehen. Ein String darf das Zeilenende nicht überschreiten.

Beispiele für Strings sind:

- `"Dies ist ein String"`
- `"Fehler in Zeile 10"`
- `"a"`
- `"xx"`

Ist `"f''(x) = 2x"` eine korrekte Zeichenkette in C? Ja, innerhalb einer Zeichenkette dürfen alle Zeichen aus dem Zeichenvorrat des jeweiligen Rechners verwendet werden, also auch Apostrophe (') sowie Umlaute (ä, ö und ü), die ansonsten in C-Code nicht vorkommen können (siehe Tabelle 2.3 auf der vorherigen Seite). Einzige Ausnahmen sind das Anführungszeichen ("), das ja das Ende des Strings kennzeichnet, und der Backslash (\). Um sie dennoch in Strings verwenden zu können, ist einfach ein Backslash davor zu setzen: `"Dies ist ein \"String\" mit Anführungszeichen"`.

In einem String können auch die oben genannten Sonderzeichen enthalten sein, wie das bereits in unserem Beispielprogramm *happy* der Fall war. In dem Ausgabestring `"Asterix ist da ! \n"` ist das Sonderzeichen *newline* enthalten.

Also nochmal: Zeichen werden zwischen Apostrophe (') gesetzt, z.B. `'A'`, Strings zwischen Anführungszeichen ("), z.B. `"Text"`. Beachten Sie dabei, dass `'a'` ein Zeichen ist, `"a"` hingegen ein String mit nur einem Zeichen, was syntaktisch im Programm eine ganz andere Bedeutung hat.

Ist die folgende Bauernregel in dieser Form ein korrekter String im Sinne von C?

```
"Wenn der Hahn kräht auf dem Mist,
ändert sich's Wetter oder es bleibt wie's ist."
```

Beinahe, nur dürfen Strings eben nicht über mehrere Zeilen gehen. Einige Compiler verarbeiten solche Strings aber dennoch. Will man einen längeren String über mehrere Zeilen schreiben, so bietet es sich an, einfach mehrere Strings hintereinander zu hängen – diese werden automatisch zusammengesetzt. Den Zeilenumbruch fügt man dann mit `"\n"` ein. Unser Beispiel würde also besser

```
"Wenn der Hahn kräht auf dem Mist,\n"
"ändert sich's Wetter oder es bleibt wie's ist."
```

lauten. Mit Strings und Zeichen werden wir uns in Datentyp `char` und Zeichenketten (Strings) nochmal beschäftigen.

2.4 Kommentare

(Siehe auch Listing 2.2.)

Zur näheren Erläuterung und besseren Nachvollziehbarkeit eines Programmteils sollte man im Programm an den entsprechenden Stellen Kommentar-Texte einfügen.

Diese Kommentare werden zwischen Kommentarklammern `/*` und `*/` eingeschlossen. Innerhalb eines Kommentars dürfen beliebige Zeichen vorkommen, wie `"`, `'` oder Umlaute; aber (Achtung!) keine weiteren `*/`, er endet dann beim ersten `*/`, der Rest liefert dann beim compilieren Fehler.

Kommentare können also nicht ineinander geschachtelt werden!

Falls Ihr Kommentar nur eine Zeile umfasst, ist es auch möglich diesen mit `//` zu beginnen. Beachten Sie hierbei, dass zur Beendigung des Kommentars kein Zeichen vonnöten ist.

Kommentare können an beliebiger Stelle des Programms stehen (außer innerhalb einer Zeichenkette, dort verlieren die Kommentarzeichen ihre Sonderbedeutung). Sie werden vom Compiler übersprungen und dienen rein der besseren Lesbarkeit und Verständlichkeit des Programmes.

So kann z.B. eine Zeile aus unserem ersten Beispiel *happy.c* auch

```
printf("\n"); /* Zeilenwechsel */

oder

printf("\n"); // Zeilenwechsel
```

heißen.

3 Datentypen und Ausdrücke

3.1 Ganzzahlige Datentypen

3.1.1 Integer

Alle in einem Programm vorkommenden Konstanten oder Variablen sind von einem bestimmten Datentyp, entweder einem der standardmäßig vorhandenen oder einem von uns selbst vereinbarten. Beginnen wir mit dem Standard-Datentyp für Integer-Werte, `int`. Dieser Typ umfasst die Menge der ganzen Zahlen. Konstanten dieses Typs sind also z.B. die folgenden Integerzahlen:

```
153   -786   0   3   -1
```

Allgemein gilt die Definition (in EBNF):

```
Integerzahl ::= [" + "|" - "] Ziffer {Ziffer}
```

Zahlen des Typs `int` kann man auch in oktaler oder hexadezimaler Form angeben. Um dies kenntlich zu machen, wird eine `0` (die Zahl Null, nicht der Buchstabe `O`) für oktal bzw. `0x` für hexadezimal vorangestellt.

Die dezimale Integerzahl 14 lautet also oktal 016 oder hexadezimal 0xE. Die führende Null bei der hexadezimalen Darstellung ist zur Unterscheidung von Namen notwendig: xE wäre ein Name! Es muss also das erste Zeichen eine Ziffer (0-9) oder ein Vorzeichen sein. Gerade die Darstellung oktaler Zahlen führt häufig zu Problemen: `08` ist gar nicht definiert (solche Fehler sind relativ leicht zu finden), aber `011` ist nicht etwa 11, sondern (weil Oktalzahl) 9.

Eigentlich ist eine Integerzahl also:

```
Integerzahl ::= [" + "|" - "] [" 0x "] Ziffer {Ziffer}
```

3.1.2 Operatoren

Es gibt nun eine Reihe von Operatoren, mit denen Daten der Integer-Typen verknüpft werden können, etwa die aus der Algebra bekannten

```
+   -   *   /   %
```

Der Operator * führt die Multiplikation aus, / ist der Divisions-Operator und % liefert den Rest der ganzzahligen Division (Modulo-Division). Dabei gelten die üblichen Prioritäten („Punkt vor Strich“), also *, / und % werden vor der Addition und Subtraktion (+,-) ausgeführt.

Sehen wir uns die beiden Operatoren % und / etwas genauer an. Die Operation `x / y` liefert als Ergebnis den ganzen Teil des Bruchs ohne Rest. Es wird also nicht gerundet, sondern es werden schlichtweg alle Nachkommastellen absgeschnitten.

Ein Beispiel:

```
17 / 3 = 5
```

Die Operation `x % y` ist nur für positive `x` und `y` definiert und hat als Ergebnis den Rest der Division, z.B.

```
20 % 3 = 2
18 % 3 = 0
```

Welches Ergebnis erhalten wir aber für 27 % 7 / 3? Man kann mit runden Klammern () , wie aus der Mathematik gewohnt, Teilausdrücke gruppieren und damit die Reihenfolge der Berechnung festlegen. Damit sind folgende zwei Zeilen identisch, nur dass die zweite deutlicher zu verstehen ist:

```
a / b % c / d
((a / b) % c) / d
```

Damit ergibt sich:

```
27 % 7 / 3
(27 % 7) / 3
6 / 3
2
```

Operatoren gleicher Priorität werden meist von links nach rechts ausgewertet. Genaueres dazu finden Sie auch im Anhang B.2.

```
#include <stdio.h>                  /* nötig für Ein−/Ausgabe */

int main()                          /* Programm−Kopf */
{
  int a, b;                         /* Deklaration von Integer Variablen */

  a = 27 % 7 / 3;                   /* a wird ein Wert zugewiesen */
  b = (27 % 7) / 3;                 /* b wird ein Wert zugewiesen */
  printf ("a = %d, b= %d\n", a, b); /* Ausgabe von a und b; siehe Kap. 4 */
```

```
    return 0;

}/* main */
```

Listing 3.1: Integer in C

In Moodle unter den ergänzenden Dokumenten zum Skript finden Sie unter Kapitel 3 oben erwähntes Beispiel erweitert zu einem kompletten Programm. Um das Programm *Integer in C* vollständig zu verstehen, empfiehlt es sich, das Kapitel 3 zuerst vollständig zu lesen. Die Ausgabe der Variablen am Bildschirm erfolgt mit dem `printf`-Befehl. Genaueres dazu finden Sie in Kapitel 4.

3.1.3 Weitere ganzzahlige Datentypen

Neben dem Standarddatentyp `int` gibt es noch den Typ `long int` (oder nur `long`) für lange Integerwerte. `int` kann sowohl eine 16 Bit, als auch eine 32 Bit Zelle oder einen Wert dazwischen aufweisen. `long` hingegen ist häufig eine 32 Bit Zelle. Dies ist im Standard nicht festgelegt, kann also von Rechner zu Rechner verschieden sein. Diese Datentypen umfassen damit also in der Praxis nicht „die Menge der ganzen Zahlen“, sondern nur den jeweils darstellbaren Bereich, d.h mit 16 Bit nur die Werte -32768, . . . , +32767.

Zu den ganzzahligen Datentypen gehört in C auch der Typ `char`, üblicherweise eine Bytezelle (8 Bit), die eigentlich zur Speicherung eines Zeichens dient (siehe Kap 9.2). Der Inhalt dieser Bytezelle ist aber gleichzeitig ein ganzzahliger Wert, mit dem man rechnen kann, der Wertebereich ist dabei auf -128 . . . +127 begrenzt. Kommen wir nun zum Datentyp `unsigned`. Die Angabe `unsigned` kann in Kombination mit allen genannten Integer-Typen verwendet werden (`unsigned long`) oder alleine stehen (nur `unsigned`, dann wird `unsigned int` angenommen). Dieser Typ umfasst alle nicht negativen ganzen Zahlen (also die natürlichen Zahlen und 0) mit den genannten Einschränkungen. Da hier ein Bit mehr zur Verfügung steht, das Vorzeichenbit, addiert sich der negative Bereich zum positiven Bereich. Bei einer Wortlänge von 16 Bit können Zahlen des Typs `unsigned`, der ja alle nichtnegativen ganzen Zahlen umfasst, maximal den Wert 65535 annehmen. Auf diese Grenzwerte muss man gegebenenfalls bei der Programmierung achten. Die arithmetischen Operatoren für diesen Typ sind dieselben, die uns für die Integertypen zur Verfügung stehen.

Welche Operatoren sind dies? Für alle diese Datentypen gelten dieselben Operatoren

```
+   -   *   /   %
```

wobei auch die Rechenregeln gleich bleiben.

Arbeitet man mit diesen Datentypen (rechnen, abspeichern), so muss man immer den möglichen Wertebereich berücksichtigen; ob bei Überlauf oder Unterlauf eine Fehlermeldung erfolgt oder nicht, ist im Standard nicht festgelegt. Das Ergebnis kann also auch ohne eine Fehlermeldung falsch sein.

In C kann man auf jedes einzelne Bit zugreifen, dies macht natürlich nur Sinn bei den ganzzahligen Datentypen (inclusive `char`). Neben den bisher bekannten arithmetischen Operatoren gibt es dafür die Bit-Operatoren.

Bitweise logische Operatoren (müssen ein Integer sein)

`&`	bitwise AND	`a & b`
`|`	bitwise OR	`a | b`
`^`	bitwise XOR (Antivalenz)	`a ^ b`
`~`	bitwise NOT	`~a`

Mit diesen Operatoren kann man also über eine Zelle beliebige Masken legen, einzelne Bits setzen oder ausblenden. Der Inhalt einer ganzen Zelle kann nach rechts oder links verschoben werden.

- `a << b` (`a` um `b` Stellen nach links schieben)
  `b` muss positiv sein, Nullen werden nachgezogen. Z.B.:

```
unsigned char a,b;
a = 3;          /* a = 0b00000011 */
b = (a << 3);   /* b = 0b00011000 */
```

- `a >> b` (`a` um `b` Stellen nach rechts schieben)
  falls `a` positiv ist, werden Nullen nachgezogen; ist `a` negativ, ist das Resultat maschinenabhängig, meist werden Nullen nachgezogen. Z.B.:

```
unsigned char a,b;
a = 240;          /* a = 0b11110000 */
b = (a >> 2);     /* b = 0b00111100 */
```

Linksschieben entspricht damit einer Multiplikation mit 2^b, *Rechtsschieben* einer Division durch 2^b.

Laut Standard gilt dies nur für positive Werte und solange der Wertebereich nicht überschritten wird. Bei negativen Werten ist das Vorzeichenbit gesetzt (höchstwertiges Bit) und es wird mit dem Komplement gearbeitet. Wenn also beim Rechtsschieben abhängig von dem Vorzeichenbit eine 1 nachgezogen wird (was allerdings nicht garantiert ist), so gilt dies auch für negative Werte.

3.1.4 Operator, Operand und Ausdruck

Wir haben nun die Begriffe Operator, Operand und Ausdruck verwendet, was bedeuten sie? In der Regel werden zwei Operanden mit einem Operator verknüpft und bilden damit einen Ausdruck. Der Ausdruck hat einen Wert, das Ergebnis dieser Rechenoperation. Ausnahmen sind z.B. die Operatoren ++ und `--`, sie können auch mit einem Operanden allein verwendet werden und bilden syntaktisch schon einen Ausdruck.

Auch der Ergebniswert eines Ausdrucks hat einen eindeutigen Datentyp. Er entspricht dem Typ der Operanden.

Ist es damit möglich, Operanden der verschiedenen Typen (int, long, unsigned) im selben Ausdruck zu verwenden? Ist also der Ausdruck

```
(i + l) * l
```

erlaubt, wenn `i` vom Typ `int` und `l` vom Typ `long` ist?

Ja, in C ist es erlaubt, Datentypen in einem Ausdruck zu mischen, es erfolgt dann eine automatische *Typumwandlung* (*type conversion*). Dies ist in anderen Programmiersprachen häufig nicht in dieser Weise erlaubt. Wenn in C eine Variable `i` vom Typ `int` und `l` vom Typ `long` ist, kann man einfach schreiben:

```
l + i
```

In diesem Fall wird `i` nach `long` konvertiert und dann in `long` gerechnet, das Ergebnis ist dann wiederum vom Typ `long`. Es gilt die Regel: Der niederwertigere Datentyp wird in den höherwertigen Typ konvertiert, dann erfolgt die Rechenoperation, das Ergebnis ist dann von diesem Datentyp. „Niederwertig“ und „höherwertig“ ist im Sinne des kleineren bzw. größeren Wertebereiches zu verstehen.

3.1.5 Ausgabe

Die Ausgabe von Integerzahlen sowie der anderen Grunddatentypen wird, ähnlich wie die Ausgabe eines Strings in unserem Beispiel *happy.c*, auch mit der Funktion `printf` durchgeführt.

Der genaue Aufruf lautet:

```
printf( "%d" , zahl);
```

Eine ausführliche Beschreibung der Funktion `printf` sowie weiterer Ein- und Ausgabefunktionen erfolgt in Kapitel 4.

Zuerst aber noch folgendes Beispiel:

```
int zahl1 = 25;
printf("Die Variable zahl1 hat den Wert %d \n", zahl1);
```

Mit der Funktion `printf` können also Text und Zahlenwerte gemeinsam ausgegeben werden.

3.2 Gleitkommazahlen

3.2.1 Datentypen float und double

Die Datentypen `float` und `double` umfassen die Menge der reellen Zahlen. Konstanten dieses Typs sind Realzahlen, die folgende Gestalt haben:

```
Realzahl ::= [" + "|" - "] Ziffer {Ziffer} " . " {Ziffer} [Zehnerpotenz] .
Zehnerpotenz ::= (" E " | " e ") [" + "|" - "] Ziffer {Ziffer} .
```

Wie sehen also die folgenden reellen Zahlen in C-Schreibweise aus?

$$2,96 \cdot 10^{-12} \quad 8,86 \cdot 10^{-19}$$

Konstanten des Typs `float` oder `double` schreibt man als gebrochene Zahl (mit Dezimalpunkt, kein Komma!), wobei man noch eine Zehnerpotenz anfügen kann (mit dem Buchstaben e oder E). Der Teil vor oder nach dem Dezimalpunkt darf auch weggelassen werden, nicht jedoch beides. Obige Zahlen schreibt man in C somit als `2.96e-12` und `8.86e-19`.

Der Unterschied zwischen `float` und `double` ist lediglich, dass für `double`-Werte ein größerer Speicherplatz im Rechner vorgesehen wird. Damit steht für Exponent und Mantisse jeweils mehr Platz zur Verfügung:

- Mehr Platz für Exponent bedeutet erweiterter Wertebereich
- Mehr Platz für Mantisse bedeutet erweiterte Genauigkeit

Um die erhöhte Genauigkeit auszunutzen, sollte möglichst `double` verwendet werden. Die mathematischen Bibliotheksfunktionen verwenden immer Argumente vom Typ `double` und liefern einen `double`-Wert zurück.

```
#include <stdio.h>     /* nötig für Ein-/Ausgabe */

int main()             /* Programm-Kopf */
{
  float a, b, c, d;    /* Deklaration von float -Variablen */

  a = 2.96e-12;        /* a wird ein Wert zugewiesen */
  b = 8.86e-19;        /* b wird ein Wert zugewiesen */
  c = .732;            /* 0,732 in kurzer C-Schreibweise */
  d = 3.;              /* 3. entspricht 3,0 */

  /* Ausgabe siehe Kap. 4*/
  printf ("a= %e, b= %e c= %f d= %f\n",a, b, c, d);
  return 0;
```

```
}/* main */
```

Listing 3.2: Float in C

In Moodle unter den ergänzenden Dokumenten zum Skript finden Sie unter Kapitel 3 das Programm *Float in C*, das die Verwendung des Typs `float` illustriert. Um das Programm vollständig zu verstehen empfiehlt es sich, das Kapitel 3 zuerst vollständig zu lesen. Die Ausgabe der Variablen am Bildschirm erfolgt mit dem `printf`-Befehl. Genaueres dazu finden Sie in Kapitel 4.

3.2.2 Typumwandlung

Implizite Typumwandlung

Gemischte Ausdrücke, d.h. Ausdrücke, mit denen Gleitkommadaten untereinander und mit Daten eines anderen Typs (z.B. `int` oder `long`) verknüpft werden, sind in C auch erlaubt. Es erfolgt eine automatische (implizite) Typumwandlung in den entsprechenden Gleitkommatyp, der ja einen größeren Wertebereich hat als ein Integer-Typ.

Betrachten wir den gemischten Ausdruck

```
5 + 7.2
```

Der Ausdruck wird wie folgt berechnet:

- Typwandlung 5 (`int`) nach 5.0 (`double`)
- Addition mit dem `double`-Wert 7.2
- Ergebnis: 12.2

Bei komplizierteren Ausdrücken ist zu beachten, dass die Umwandlung immer für jeden Einzel-Ausdruck einzeln stattfindet.

Welches Ergebnis hat also der Ausdruck (5/2) * 2.2 + 1? Zuerst wird der Teil-Ausdruck `5/2` berechnet. Da hier beide Operanden vom Typ `int` sind, wird auch mit Integer gerechnet. Das Ergebnis der Klammer ist also 2. Vor der Multiplikation mit 2.2 findet dann aber die Umwandlung nach `double` statt, das Zwischenergebnis ist 4.4. Bei der abschließenden Addition wird ebenfalls in `double` gerechnet, das Ergebnis des Ausdrucks ist also 5.4.

Bei einigen Typumwandlungen gibt der Compiler eine Warnung aus, erzeugt aber dennoch den entsprechenden Code.

Explizite Typumwandlung

Es existiert jedoch auch die Möglichkeit der expliziten Typwandlung unter Verwendung des Cast-Operators:

```
(neuer Typ) wert
```

liefert den Wert konvertiert in den neuen Datentyp.

Also liefert `(float)wert` oder `(double)wert` die Variable `wert` als Gleitkommazahl, umgekehrt liefert `(int)x` den ganzzahligen Teil der Gleitkommazahl x.

Die Typwandlung `(int)x` wandelt eine Gleitkommazahl in eine Integerzahl um, indem einfach alles, was nach dem Dezimalpunkt steht, abgeschnitten wird. Es erfolgt also keine Rundung. Dies gilt für positive und für negative Werte.

Welches Ergebnis liefert die Typwandlung (int)x für x=1.96? Da die Nachkommastellen abgeschnitten werden ergibt sich 1. Aus `3e-2` wird 0.

3.2.3 Operatoren

Für die Gleitkomma-Datentypen stehen uns die gleichen Operatoren zur Verfügung

```
*   /   +   -
```

aber natürlich keine Modulo-Operation (%).

Beachten Sie bitte, dass der Cast-Operator eine höhere Priorität besitzt als die arithmetischen Operatoren.

3.3 Boolesche Werte

Einen logischen Datentyp, der nur die beiden Werte TRUE und FALSE annehmen kann, den Sie vielleicht aus anderen Programmiersprachen kennen, gibt es in C nicht. Hier wird ersatzweise `int` verwendet, wobei 0 dem Wert FALSE entspricht und ein beliebiger Wert ungleich 0 (z.B. 1) dem Wert TRUE.

Mehr zu logischen Operatoren und Ausdrücken in Kapitel 5.

3.4 Variablenvereinbarung, Vorbelegung, Konstante

3.4.1 Variablenvereinbarung

In C müssen alle im Programm vorkommenden Variablen zu Beginn vereinbart werden, d.h. mit ihrem Namen und ihrem Typ aufgeführt sein. Das sieht dann z.B. so aus:

```
double x, y, z;   /* Variablenvereinbarungen     */
int nummer, i, j; /* Typ Variablenliste          */
long l;           /* Semikolons nicht vergessen! */
```

Variablen desselben Typs können, durch Kommata getrennt, gemeinsam als Liste vereinbart werden. Die Variablenvereinbarungen stehen am Beginn des Programms im sogenannten Vereinbarungsteil.

Von welchem Typ ist also in unserem Fall die Variable y? Sie befindet sich in der ersten Zeile, in der hier Variablen vom typ `double` vereinbart werden.

Sehen wir uns folgenden Vereinbarungsteil eines anderen Programms an:

```
int a, b, c;
float preis;
unsigned nr;
char zeichen;
```

Die Variablen `a`, `b` und `c` sind hier vom Typ `int`, die Variable `preis` ist ein Gleitkommawert. Die Variable `nr` kann alle natürlichen Zahlen oder die 0 annehmen. Die Variable `zeichen` ist vom Typ `char`. Dieser Datentyp dient zum Abspeichern eines Zeichens.

Ist folgender Vereinbarungsteil korrekt?

```
float r, f; double hoehe,laenge; int x,y;
```

Natürlich ist es unerheblich, wieviele Leerzeichen zwischen den einzelnen Zeichen stehen, nur nicht innerhalb eines Namens. C ist völlig formatfrei, d.h. man kann Vereinbarungen wie auch Anweisungen beliebig über die Zeilen verteilen oder mehrere Vereinbarungen in eine Zeile schreiben. Man sollte sich aber angewöhnen, die Programme so strukturiert zu schreiben, dass sie auch für fremde Benutzer übersichtlich, lesbar und verständlich sind. Für den Rechner, genauer gesagt für den Compiler, der das Programm in Maschinensprache übersetzt, ist die Form bedeutungslos. Hier ist entscheidend, dass die Syntax der Sprache eingehalten wird und damit verständlich ist. Dies gilt natürlich nicht nur für den Vereinbarungsteil.

Namen wurden bereits in Kapitel 2.3.3 besprochen. Sie können Buchstaben und Ziffern enthalten, aber keine Sonderzeichen und auch keine Leerzeichen! Das erste Zeichen muss ein Buchstabe sein. Zur besseren Lesbarkeit der Programme sollten sinnvolle Namen vergeben werden. Es hat sich als guter Stil eingebürgert, nur Kleinschreibung zu verwenden.

Wie lange darf nun ein solcher Name sein?

Nach dem ANSI Standard darf ein Name beliebig lang sein, aber maximal 31 Zeichen sind signifikant (d.h. werden zur Unterscheidung zweier Namen auch tatsächlich benutzt, der Rest wird einfach ignoriert).

Mit der Vereinbarung einer Variablen wird festgelegt, dass in dem Programm oder Programmteil eine Speicherzelle verwendet werden soll, die einen bestimmten Namen haben soll, damit man sie im Programm verwenden kann. Der verwendete Name sollte also sinnvoll gewählt werden, damit das Programm lesbar wird. Gleichzeitig wird gesagt, welcher Typ von Daten in dieser Speicherstelle abgelegt werden kann.

Je nach Datentyp kann die Speicherstelle unterschiedliche Größe haben. Für ein Zeichen genügt ein Byte, für eine doppelt genaue Gleitkommazahl werden üblicherweise 8 Bytes verwendet. Der Inhalt dieser Speicherzelle ist damit aber noch undefiniert, sie

kann also ein beliebiges zufälliges Bitmuster enthalten, das damit einen völlig unsinnigen Wert darstellen kann.

3.4.2 Vorbelegung

Soll die Variable bereits zu Beginn einen sinnvollen Wert enthalten, so muss sie bei der Vereinbarung zusätzlich vorbelegt werden, was als Initialisierung bezeichnet wird. Soll also die Integer-Variable `i` zunächst den definierten Wert 1 annehmen, so schreibt man

```
int i = 1;
```

Dieser bei der Initialisierung angegebene Wert ist nur ein Startwert, er kann natürlich später im Programm beliebig verändert werden. Auf die gleiche Weise kann z.B. die Variable `messwert` vorbelegt werden:

```
double messwert = 123.456;
```

In der Zelle `i` vom Typ `int` wird ein ganzzahliger Wert abgelegt, in der Zelle `messwert` ein Gleitkommawert.

Sind die folgenden Initialisierungen richtig?

```
int zaehler=0, j=10, k;
float x=100., y, z;
char zeichen='a';
double pi;
```

Ja, alle Variablen sind korrekt definiert. Die Variable `zaehler` ist mit 0 vorbelegt, `j` mit 10 und `x` mit 100.0. Der Inhalt von `k`, `y` und `z` ist noch undefiniert. Auch die `char`-Variable `zeichen` ist mit dem Buchstaben a vorbelegt, während der Inhalt der Zelle mit dem Namen `pi` undefiniert ist.

Der Name `pi` ist für den Rechner nichts besonderes, er muss also auch vorbelegt werden. Soll sie den Wert von π erhalten, könnte man z.B. schreiben:

```
double pi = 3.14159;
```

3.4.3 Konstanten

Der Inhalt von `pi` könnte damit aber im Programm verändert werden, was in diesem Fall wohl nicht erwünscht ist. Durch das zusätzliche Keyword `const` kann dies verhindert werden, die Variable `pi` wird zu einer Konstanten.

```
const double pi = 3.14159;
```

Konstanten müssen bei der Vereinbarung initialisiert werden, Zuweisungen an Konstanten sind nicht erlaubt.

3.4.4 Initialisierung und Zuweisung

Noch ein wichtiger Hinweis: Eine Initialisierung ist etwas anderes als eine Zuweisung, zwischen diesen beiden Begriffen wird in C unterschieden, so dass z.B.

```
int i = 1; /* Vereinbarung und Initialisierung mit dem Wert 1 */
```

streng genommen nicht dasselbe ist wie

```
int i;    /* Vereinbarung ohne Initialisierung */
i = 1;    /* Zuweisung des Wertes 1 */
```

auch wenn es bei einfachen Datentypen – wie hier `int` – dasselbe bewirkt.

Bei komplizierteren Datentypen wie etwa Zeichenketten, die Sie in Kapitel 8 kennenlernen werden, macht es jedoch durchaus einen Unterschied, ob man eine Variable mit einem Startwert initialisiert oder ihr einen Wert zuweist. Machen Sie sich deshalb bereits jetzt den Unterschied zwischen beiden bewusst.

Eine Initialisierung kann nur gleichzeitig mit der Vereinbarung der Variablen erfolgen.

3.5 Wertzuweisung, Ausdrücke und Anweisungen

3.5.1 Wertzuweisung

Ein wichtiger syntaktischer Begriff ist der Begriff *Anweisung*. Anweisungen beschreiben die Aktionen, die mit bestimmten Daten durchgeführt werden sollen. Eine einfache Anweisung ist die Wertzuweisung. Dabei wird der momentane Wert einer Variablen durch einen neuen Wert ersetzt.

Beispiele:

```
i = 0;      /* Symbol "=" für Wertzuweisung     */
a = b + c;  /* man kann auch das Ergebnis von Ausdrücken zuweisen */
```

Die Variablen `a`, `b` und `c` aus dem obigen Beispiel seien vom Typ `int`. Sie wollen nun der Variablen `c` den Wert der Division „a dividiert durch b“ zuweisen. Wie sieht das aus?

Die gefragte Wertzuweisung sieht so aus:

```
c = a / b
```

Allgemein formuliert (EBNF) hat die Wertzuweisung die Form

```
Wertzuweisung ::= Variable " = " Ausdruck .
```

Ein Ausdruck kann dabei ein beliebig komplexer Ausdruck, aber auch eine einzelne Variable oder Konstante sein.

Den Begriff Ausdruck haben wir schon kennengelernt. Ein arithmetischer Ausdruck wird entsprechend den Regeln berechnet, dabei findet in C bei unterschiedlichen Datentypen der Operanden eine automatische Typumwandlung statt. Der Wert, das Ergebnis des Ausdrucks, hat einen bestimmten Datentyp. Auch bei der Zuweisung erfolgt bei unterschiedlichen Typen in C eine automatische Typwandlung. Der Wert des Ausdruckes wird in den Typ der (linksstehenden) Variablen umgewandelt.

Das Ergebnis des (rechts) berechneten Ausdruckes wird der Variablen (links) zugewiesen, der alte Inhalt der Variablen wird dabei überschrieben.

Betrachten wir den Ausdruck

```
i = i + 1
```

Dies ist keine Gleichung, wie es auf den ersten Blick erscheinen könnte, sondern eine Zuweisung. In der Algebra wäre diese Gleichung falsch. In einem Programm ist dies aber ein erlaubter und auch sinnvoller Ausdruck.

Zunächst wird der Ausdruck

```
i + 1
```

berechnet, das Ergebnis wird der Variablen `i` wieder zugewiesen, wobei der alte Inhalt überschrieben wird. Der Inhalt der Zelle `i` wurde um 1 erhöht.

In Kapitel 3.5.5 werden wir noch eine andere Möglichkeit kennenlernen, den Inhalt einer Zelle um 1 zu erhöhen. Sehen wir uns vorher aber noch ein paar Beispiele an.

Wir haben in einem Programm folgenden Vereinbarungsteil:

```
float z = 14.2, y = 28.4, x;
int i = 5, j = 23, k;
```

Welches Ergebnis liefern folgende Zuweisungen?

- `x = z / y`
  liefert in `x` das Ergebnis 0.5, hier ist keine Typwandlung erforderlich
- `k = z / y`
  liefert in `k` das Ergebnis 0; das Ergebnis des Ausdrucks `z/y` ist 0.5; da es in den Typ der Variablen, also nach `int`, gewandelt werden muss, bleibt nur der ganzzahlige Teil, also 0. Die Nachkommastellen werden gestrichen.
- `k = i / j`
  liefert in `k` das Ergebnis 0, da die Integer-Division `5 / 23` den ganzzahligen Wert 0 liefert.
- `x = i / j`
  liefert in `x` ebenfalls 0.0, das Ergebnis der Division ist 0, erst dann erfolgt die Wandlung nach Gleitkomma 0.0

In C ist die Zuweisung auch ein Operator, damit ist eine Zuweisung syntaktisch auch ein Ausdruck. Somit kann man sogar folgendes schreiben:

```
k = x = i + j;
```

Es wird zunächst die Zuweisung `x = i + j` ausgeführt; das Ergebnis dieser Zuweisung (hier 28) wird dann der Variablen `k` zugewiesen.

```
#include <stdio.h>          /* nötig für Ein-/Ausgabe */

int main()                  /* Programm-Kopf */
{
  float z=14.2, y=28.4, x;  /* Deklaration von Variablen */
  int i=5, j=23, k;

  x = z / y;                /* Zuweisung mit float-Variablen */
  printf ("x=%f\n", x);     /* Ausgabe siehe Kap. 4 */

  k = z / y;                /* Achtung: k ist int */
  printf ("k=%d\n", k);     /* Ausgabe siehe Kap. 4 */

  k = i / j;                /* alle Variablen sind vom Typ int */
  printf ("k=%d\n", k);     /* Ausgabe siehe Kap. 4 */

  x = i / j;                /* Bruch besteht aus int-Variablen */
  printf ("x=%f\n", x);     /* Ausgabe siehe Kap. 4 */
  return 0;
}/* main */
```

Listing 3.3: Wertzuweisung

In Moodle unter den ergänzenden Dokumenten zum Skript finden Sie unter Kapitel 3 das Programm *Wertzuweisung*. Um das Programm vollständig zu verstehen, empfiehlt es sich das Kapitel 3 zuerst vollständig zu lesen. Die Ausgabe der Variablen am Bildschirm erfolgt mit dem `printf`-Befehl. Genaueres dazu finden Sie in Kapitel 4.

3.5.2 Ausdrücke und Anweisungen

Ein C-Programm besteht aus mehreren Teilen, wie wir schon an unserem Beispiel *happy.c* gesehen haben:

```
#include <stdio.h>          /* Einfügungen */

int main()                  /* Programm-Kopf */
{
  printf ("Hurra, hurra!");  /*        -        */
  printf ("\n");             /* Programm-Block */
```

```
    printf ("Asterix ist da!\n");  /*        –        */
    return 0;                      /*        –        */
}/* End Happy */                   /* Kommentar      */
```

Der eigentliche Programm-Block besteht aus einer oder mehreren Anweisungen, die in der gegebenen Reihenfolge abgearbeitet werden. Wie wird nun aus einem Ausdruck, den wir gerade kennengelernt haben, eine Anweisung?

An einen Ausdruck wird einfach ein Strichpunkt angehängt.

- Ausdruck: `c = a / b`
- Anweisung: `c = a / b;`

Allgemein:

```
Anweisung ::= Ausdruck ";" .
```

Anweisungen können aneinandergereiht werden zu einer Anweisungsfolge, so wie sie in einem Programm abgearbeitet werden sollen:

```
Anweisungsfolge ::= Anweisung {Anweisung} .
```

Ich möchte die Werte der Variablen `a` und `b` vertauschen, d.h. `a` soll den Wert von `b` bekommen und umgekehrt, und schreibe deshalb:

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

Hat diese Anweisungsfolge den gewünschten Effekt? Nein. Die Anweisungsfolge leistet keine Vertauschung der Werte von `a` und `b`. Bei der Wertzuweisung erhält die Variable `a` einen neuen Wert und der alte Wert geht verloren. Die obige Anweisungsfolge führt also zu dem Ergebnis, dass beide Variablen den gleichen Wert haben, nämlich den Wert von `b`. Um eine Vertauschung zu erreichen, benötigt man eine Hilfsvariable, z.B. die Variable `c`.

Welche der Anweisungsfolgen führen zum Erfolg?

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

In `(a)` geht in der ersten Anweisung der Wert von `a` verloren. Der Hilfsvariablen wird dann der neue Wert von `a`, der identisch mit `b` ist, zugeweisen.

Die zweite Version `(b)` führt zum Erfolg: Der Wert von `a` wird in `c` zwischengespeichert, dann wird `a` mit `b` überschrieben, und schließlich der gesicherte Wert an `b` zugewiesen.

Die letzte Version `(c)` verliert den Wert von `a`.

Haben die folgenden beiden Zeilen den gleichen Effekt?

```
a = a + 1; b = 4 * a;
b = 4 * a; a = a + 1;
```

Nein, auch hier muss wieder die Reihenfolge der Anweisungen beachtet werden.

3.5.3 Leere Anweisung

Eine Anweisung kann auch aus allein dem Strichpunkt ; bestehen. Sie heißt dann leere Anweisung und führt keine Aktion aus. Die leere Anweisung gibt es vorwiegend aus syntaktischen Gründen: man darf dadurch Semikolons an Stellen einfügen, wo sie eigentlich überflüssig wären.

Es gibt aber auch sinnvollere Anwendungen, z.B. im Zusammenhang mit Schleifen.

3.5.4 Ausdruck

Wir haben schon desöfteren den Begriff Ausdruck verwendet. Diesen Begriff wollen wir uns jetzt etwas näher ansehen.

Ein Ausdruck setzt sich zusammen aus verschiedenen Operanden und Operatoren (vgl. Kapitel 3.1.4). Die Operanden können Konstanten, Variablen oder Funktionen sein (zum letzten Begriff mehr in Kapitel 10.1.1). Die Operatoren haben eine feste Rangordnung, die wir aus der Algebra kennen. Die Rangordnung legt die Reihenfolge fest, in der ein umfangreicherer Ausdruck abgearbeitet wird.

Beispiele für Ausdrücke sind

`8 * 3 + 7 * 4`	$(= (8 \cdot 3) + (7 \cdot 4) = 52)$
`8 + 3 * 7 + 4`	$(= 8 + (3 \cdot 7) + 4 = 33)$
`19 / 6 * 2`	$(= (19/6) \cdot 2 = 6)$
`19 / (6 * 2)`	$(= 19/12 = 1)$.

Für die Abarbeitung von Ausdrücken gelten also analog die Regeln der Algebra. Bei Ausdrücken werden die Operationen in dieser Reihenfolge ausgeführt:

1. Die Operationen, die innerhalb von runden Klammern stehen (...)
2. dann der Negationsoperator: -
3. die Multiplikationsoperatoren: * / %
4. die Additionsoperatoren: + - und zuletzt
5. die Zuweisungsoperatoren: =

Weitere Operatoren werden wir später noch kennenlernen und in diese Prioritätenliste aufnehmen (vgl. Kapitel 5.4).

Welches Ergebnis liefert der folgende Ausdruck?

```
16 % ((5 * 2) + 5)
```

Zuerst werden die Klammerausdrücke ausgeführt, dann der Modulo-Operator. Es ergibt sich also `16 % (10 + 5) = 16 % 15 = 1`. Es waren übrigens gar nicht alle verwendeten Klammern erforderlich, aber zur besseren Lesbarkeit (oder wenn Sie die Prioritätenliste nicht parat haben) dürfen (und sollten!) natürlich überflüssige Klammern verwendet werden.

3.5.5 Inkrement und Dekrement

Weitere arithmetische Operatoren sind:

- Inkrement (++)
- Dekrement (--)

Das bedeutet, der Operand wird um 1 erhöht bzw. um 1 verringert. Sie gehören zur Gruppe der *unary*-Operatoren, d.h. sie werden nur mit einem Operanden verwendet.

- `i++` oder `++i` ist gleichbedeutend mit `i = i + 1`
- `i--` oder `--i` ist gleichbedeutend mit `i = i - 1`

Die Verwendung des Post-Inkrements `i++` bzw. des Prä-Inkrements `++i` hat nur in einem umfangreicheren Ausdruck eine unterschiedliche Bedeutung:

- bei Post-Inkrement (`i++`) wird der Operand `i` zuerst verwendet und dann inkrementiert
- bei Prä-Inkrement (`++i`) wird der Operand `i` zuerst inkrementiert und dann verwendet

Das gleiche gilt natürlich auch für den Dekrement-Operator.

Welche Werte haben die vier Variablen `x`, `y`, `a` und `b` am Ende der folgenden Zeilen?

```
int x=1, y=5, a, b;
a = x++;
b = --y;
```

In der ersten Zeile werden die Variablen vereinbart und zum Teil auch initialisiert. In der zweiten Zeile wird der Variablen `a` ein Wert zugewiesen. Da hier bei `x` ein Post-Inkrement Operator steht, wird `x` zuerst verwendet, und dann um eins erhöht. Das bedeutet, dass `a` der ursprüngliche Wert von `x` zugewiesen wird, nämlich 1. Dann erst wird `x` erhöht und erhält den Wert 2.

In der letzten Zeile wird `b` ein neuer Wert zugewiesen. Hier findet ein Prä-Dekrement statt, also wird `y` zunächst um 1 verringert und erhält den Wert 4. Dieser neue Wert wird dann verwendet und `b` zugewiesen.

Auch die Inkrement- und Dekrement-Operatoren können auf alle Datentypen angewendet werden. Sie werden in der Priorität vor den multiplikativen bzw. additiven Operatoren ausgeführt. Die Rangordnung der bisher bekannten Operatoren in abfallender Reihenfolge sieht also wie folgt aus:

1. Unary: `++ --`
2. Multiplikativ: `* / %`
3. Additiv: `+ -`

Ein Potenzierungsoperator existiert in C nicht, dies muss mit einer Multiplikation oder mit Hilfe einer Bibliotheksfunktion erledigt werden.

```
#include <stdio.h>              /* nötig für Ein-/Ausgabe */

int main()                      /* Programm-Kopf */
{
  int x=1, y=5, a, b;
  a = x++;                      /* Post-Inkrement */
  b = --y;                      /* Prä-Dekrement */

  /* Ausgabe */
  printf ("a = %d, b = %d, x = %d, y = %d\n", a, b, x, y);
  return 0;

}/* main */
```

Listing 3.4: Inkrement und Dekrement

In Moodle unter den ergänzenden Dokumenten zum Skript finden Sie unter Kapitel 3 das zuletzt gezeigte Beispiel in Form des Programmes *Inkrement und Dekrement*. Die Ausgabe der Variablen am Bildschirm erfolgt mit dem `printf`-Befehl. Genaueres dazu finden Sie in Kapitel 4.

3.5.6 Regeln für Ausdrücke

Allgemein sind für Ausdrücke folgende Regeln zu beachten:

- *Alle verwendeten Variablen müssen vereinbart (deklariert) werden*
- Jeder vorkommenden Variablen muss vorher ein Wert zugewiesen worden sein
- Das Multiplikationszeichen darf nicht weggelassen werden: nicht `2n` sondern `2 * n`
- Die Operationen, die in runden Klammern stehen, werden zuerst ausgeführt
- Die Reihenfolge der Abarbeitung von komplexeren Ausdrücken ist durch die Priorität der Operatoren geregelt
- Zuletzt wird die Zuweisung ausgeführt
- Operatoren der gleichen Art werden von links nach rechts ausgeführt, nur Zuweisungen und unäre Operatoren werden von rechts nach links ausgeführt

3.5.7 Weitere Beispiele

Sehen wir uns noch ein paar Beispiele an. Die Variablen `a` und `b`, sowie `x`, `y` und `z` sind folgendermaßen vereinbart:

```
int   a, b;
float x, y, z;
```

Untersuchen Sie, welche der folgenden Ausdrücke nach der C-Syntax nicht in Ordnung sind.

```
(x - 2y) / 16           (a)
x = (a / b) / (a + b)   (b)
a + b = 15 / 4          (c)
x % y + 14 * x          (d)
a % b % b               (e)
```

In den Zeilen a fehlt der Multiplikationsoperator zwischen `2` und `y`. In Zeile c steht links des Zuweisungsoperators ein Ausdruck, an den keine Zuweisung möglich ist. Und in Zeile d wird der Modulooperator auf Gleitkommazahlen angewandt, obwohl er nur für Ganzzahlen definiert ist.

Untersuchen Sie, ob die folgenden Ausdrücke wirklich zu den angegebenen Ergebnissen führen:

```
2 + 4.25              (a) = 6,25
(-4) * 3 / 5 - 3      (b) = -6
27 / 3 % 2            (c) = 1
27 / (3 % 2)          (d) = 27
```

Die Zeile (b) ist falsch.

3.5.8 Zusammenfassung

Wir haben gelernt, dass alle Variablen, die im Programm verwendet werden, vor den Anweisungen vereinbart werden müssen. Diese Vereinbarung bedeutet, dass eine Zelle, ein Platz im Arbeitsspeicher reserviert wird, um darin einen Wert (Zahlenwert oder Zeichen) abzulegen. Also muss der Typ der Variablen definiert werden, diese Vereinbarung gilt dann für das gesamte Programm (bzw. Modul), z.B.

```
int a;
```

Nach dieser Vereinbarung ist lediglich der benötigte Platz reserviert, der Inhalt der Zelle ist undefiniert. Erst während der Ausführung des Programms (zur Laufzeit, auch Runtime) wird in der Zelle ein bestimmter Wert abgelegt. Dies geschieht in der Regel durch eine Zuweisung, z.B.

```
a = Ausdruck;
```

Dabei wird das Ergebnis des berechneten Ausdrucks in der Zelle `a` abgelegt. Der einfachste Ausdruck besteht aus einem Zahlenwert, z.B.

```
a = 123;
```

Wenn der Typ der Variablen mit dem Typ des Wertes nicht übereinstimmt, so wird automatisch eine entsprechende Typumwandlung durchgeführt.

Ein Ausdruck kann zwar auch berechnet werden, wenn den verwendeten Variablen kein Wert zugewiesen wurde, nur ist dann das Ergebnis nicht vorhersagbar: Was auch immer zufällig in der jeweiligen Speicherzelle steht, wird dann im Ausdruck verwendet. Der Compiler kann an Stellen, an denen er einen solchen Fehler vermutet, eine Warnung ausgeben, aber das Programm dennoch übersetzen.

Eine weitere Möglichkeit, um den Inhalt einer Variablen definiert zu füllen, ist durch Einlesen von Werten (z.B. über `scanf`). Dazu mehr im nächsten Kapitel 4.

4 Ein- und Ausgabe

Was nützt das schönste Programm, wenn die Ergebnisse einer Berechnung nicht ausgegeben werden. Ein Programm ist wertlos, wenn es keine Daten einlesen und ausgeben kann. In C wird die Ein-/Ausgabe durch Bibliotheksfunktionen ausgeführt. Um sie zu benutzen, muss der Header *stdio.h* eingebunden werden:

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

4.1 Ausgabe mit `printf`

Im Rechner werden die Zahlenwerte in einer Zelle als Bitmuster, als Dualzahlen abgelegt. Zur Ausgabe auf dem Bildschirm muss dieses Bitmuster nach einer Vorschrift in die entsprechenden Dezimalziffern – genauer gesagt: in die auszugebende Zeichenfolge – umformatiert werden. In einer Formatangabe wird die passende Umwandlungsvorschrift angegeben.

Da die Formatangabe die Ausgabe des internen Bitmusters in ein lesbare Zeichen steuert, muss sie zum Datentyp der Zahl passen. Hier sehen Sie eine Liste der verschiedenen Formatangaben:

`%d` oder `%i`	vorzeichenbehaftete Integerzahl
`%x`	hexadezimale Integerzahl
`%o`	oktale Integerzahl
`%u`	vorzeichenlose Integerzahl
`%ld`	`long` Zahl
`%lu`	`unsigned long` Zahl
`%f`	Gleitkommazahl (`float` und `double`)
`%e`	Gleitkommazahl in Exponentialdarstellung
`%c`	einzelnes Zeichen (`char`)
`%s`	Zeichenkette (String)

Tabelle 4.1: Formatangaben printf

Beispiele:

```
int zahl1 = 123;
unsigned zahl2 = 8698;
float fzahl = 12.34567;
```

```
printf("%d", zahl1);
printf("%u", zahl2);
printf("%f", fzahl);
```

Damit werden die Werte der einzelnen Zahlen richtig auf dem Bildschirm ausgegeben. Wählt man ein falsches Format wie z.B. %d bei dem float-Wert fzahl, so gibt es bei den meisten Compilern leider keine Fehlermeldung oder Warnung, stattdessen wird ein völlig unsinniger Wert ausgegeben, da das Bitmuster von fzahl durch die falsche Formatangabe als Integerwert interpretiert wird.

Bei der Ausgabe kann zusätzlich jeweils eine Angabe der Mindestzahl an Stellen erfolgen: %<stellenzahl>d , wobei <stellenzahl> eine ganzzahlige Konstante ist.

```
printf("%7d",zahl1);
```

<stellenzahl> gibt an, wieviele Stellen (Zeichen) mindestens auf dem Bildschirm ausgegeben werden sollen. Hier hat zahl1 den Wert +123, also werden mit

```
printf("%d",zahl1);
```

nur die benötigten 3 Zeichen 123 ausgegeben, mit

```
printf("%7d",zahl1);
```

werden 7 Zeichen ausgegeben, d.h. es wird mit führenden 4 Leerzeichen aufgefüllt, also " 123".

Die Angabe der Stellenzahl ist bei allen Datentypen möglich. Bei Gleitkommawerten kann zusätzlich die Anzahl der Nachkommastellen mit angegeben werden. Diese ist per default 6, kann aber durch Angabe einer weiteren Konstanten gesteuert werden:

```
%<stellenzahl>.<stellenzahl>f
```

Beispiele:

```
printf("%f\n",fzahl);
printf("%e\n",fzahl);
printf("%7.2f\n",fzahl);
```

Welche Ausgabe erfolgt in den drei Bespielen, wenn fzahl=12.34567 **(siehe oben) ist?** Folgende Zeilen werden ausgegeben:

```
12.345670
1.234567e+001
  12.35
```

Hier wurde durch Ausgabe des Sonderzeichens '\n' ein Zeilenvorschub erzeugt, es wird also an den Anfang der nächsten Ausgabezeile gesprungen. Der Zeilenvorschub kann in einer eigenen Anweisung,

```
printf("\n");
```

zusammen mit der Ausgabe von Zahlenwerten

```
printf("%d\n", zahl);
```

oder zusammen mit einer Textausgabe

```
printf("Es wurde der Wert %d berechnet.\n", zahl);
```

erscheinen.

```
#include <stdio.h>          /* nötig für Ein-/Ausgabe */

int main()                  /* Programm-Kopf */
{
  int zahl, zahl1=123;
  unsigned zahl2 = 8698;
  float fzahl = 12.34567;

  printf ("Ausgabe der integer-Zahl: ");
  printf ("%d", zahl1);
  printf ("\n");
  /* \" erzeugt ein Anführungszeichen */
  printf ("Ausgabe der \"unsigned integer\": ");
  printf ("%u", zahl2);
  printf ("\n");
  printf ("Ausgabe der float-Zahl: ");
  printf ("%f", fzahl);
  printf ("\n");
  printf ("7-stellige Ausgabe der integer-Zahl: ");
  printf ("\n");
  printf ("%7d", zahl1);
  printf ("\n");
  printf ("1234567");
  printf ("\n");
  zahl = zahl1 / 2;
  printf ("Es wurde der Wert %d berechnet.\n", zahl);
  printf ("\n\nZeilenvorschub\nan\nbeliebiger Stelle\n!!!\ n");
  return 0;
}/* main */
```

Listing 4.1: Ausgabe mit printf

In Moodle unter den ergänzenden Dokumenten zum Skript werden unter Kapitel 4 einige der angesprochenen Ausgabe Möglichkeiten im Programm *Ausgabe mit printf* aufgezeigt.

4.2 Eingabe mit `scanf`

Da ein Programm nicht nur mit den fest im Quellcode enthaltenen Zahlenwerten rechnen, sondern auch Werte einlesen soll, benötigen wir eine weitere I/O Funktion: `scanf` (formatiertes Einlesen). Auch diese Funktion ist in *stdio.h* beschrieben, mit der Einfügung `#include <stdio.h>` kann sie verwendet werden.

Auch hier nun nochmal die verschiedenen Formatangaben als Übersicht:

`%d` oder `%i`	vorzeichenbehaftete Integerzahl
`%x`	hexadezimale Integerzahl
`%o`	oktale Integerzahl
`%u`	vorzeichenlose Integerzahl
`%ld`	`long` Zahl
`%lu`	`unsigned long` Zahl
`%f`	Gleitkommazahl (`float`)
`%lf`	Gleitkommazahl (`double`)
`%e`	Gleitkommazahl in Exponentialdarstellung
`%c`	einzelnes Zeichen (`char`)
`%s`	Zeichenkette (String)

Tabelle 4.2: Formatangaben scanf

Leider ist die Anwendung der Eingabefunktion in C nicht ganz so einfach wie in anderen Programmiersprachen. In C kann die Lesefunktion nicht einfach in einer Variablen einen Wert ablegen, hier muss bei allen Datentypen (außer Strings) mit der Adresse und dem Adressoperator `&` gearbeitet werden.

In unserem Programm ist die Variable `int izahl` vereinbart. Dann ist die Adresse dieser Zelle `&izahl`, die in der Einlesefunktion verwendet werden muss:

```
if(scanf("%d", &izahl)==0)
{
        printf ("Unzulaessige Eingabe!");
        return 1;
}
```

Hierbei ist es wichtig, dass die scanf-Funktion in eine if-Anweisung (s.Kapitel 5 und 6) verpackt wird, da hierdurch überprüft wird, ob eine korrekte Eingabe erfolgt ist. Es wird zum Beispiel geprüft, ob wirklich ein `integer` und nicht ein `char` übergeben wurde. Falls ein `char` übergeben wurde, erhält man als Ausgabe den Text *Unzulässige Eingabe!*. Dies ist unverzichtbar für einen guten Programmierstil und sollte unbedingt beachtet werden. Der Befehl `return 1` bricht das Programm direkt nach einer falschen Eingabe ab.

Wichtig ist, dass bei `scanf` der Adressoperator `&` vor den Variablennamen gesetzt werden muss. Die Eingabe eines Zahlenwertes erfolgt, während das Programm läuft,

über die Tastatur. Nach der Eingabe eines Zahlenwertes muss ein <CR> eingegeben werden.

Nachdem die gewünschten Berechnungen durchgeführt wurden, lassen wir das Ergebnis auf dem Bildschirm ausgeben, was im Programm selbst erfolgen muss, da natürlich keine automatische Ausgabe erfolgt.

Bei der Eingabe darf im Kontrollstring kein Text enthalten sein, auch die Angabe einer Stellenzahl ist mit Vorsicht zu benutzen.

Wenn ein Programm irgendwelche Werte einliest, so müssen die betreffenden Werte dann eingegeben werden, wenn das Programm läuft, nicht etwa im Quelltext. Es ist dabei ratsam, im Programm zunächst eine Ausgabe – etwa

```
"Bitte geben Sie den Wert für X ein: "
```

– zu machen, damit der Benutzer die gewünschten Werte eingeben kann.

Es ist auch immer sinnvoll, eingelesene Werte zusammen mit den Ergebnissen der Berechnung wieder auszugeben. Damit hat man eine Kontrolle, ob das Einlesen richtig funktioniert hat, und man weiß auch, mit welchen Eingabewerten die Berechnungen erfolgten.

Sehen Sie sich das Beispiel *Eingabe mit scanf* an.

```
#include <stdio.h>      /* nötig für Ein-/Ausgabe */

int main()              /* Programm-Kopf */
{
  int i=22;
  float f=15.26;
  double d;

  printf ("Hurra, hurra.\n");
  printf ("Dies ist  eine Integer-Zahl: ");
  printf ("%5d\n", i);
  printf ("Dies ist  eine float -Zahl: ");
  printf ("%10f\n", f );
  printf ("Gib Integer -> ");
  if (scanf ("%d",&i)==0)
  {
      printf ("Unzulaessige Eingabe!\n");
      return 1;
  }
  printf ("Die Zahl war: %10d\n",i);
  printf ("Gib float  -> ");
  if (scanf ("%f", &f)==0)
  {
      printf ("Unzulaessige Eingabe!\n");
      return 1;
  }
```

```
    printf ("Die Zahl war: %15.2f\n", f);
    printf ("Das war's – Test ENDE\n");
    return 0;
}/* main */
```

Listing 4.2: Eingabe mit scanf
In Moodle unter den ergänzenden Dokumenten zum Skript finden Sie unter Kapitel 4 das Beispiel als *Eingabe mit scanf*.

4.3 Häufige Fehler

Vergessen Sie nicht bei der Programmausgabe (`printf`) einen Zeilenumbruch (*Newline*) einzufügen! In der C-Bibliothek ist das sog. *line buffering* üblich, d.h. die ausgegebenen Zeichen werden zwischengespeichert und erst angezeigt, wenn eine Zeile durch ein Newline abgeschlossen wurde (kann aber auch durch spezielle Kommandos erzwungen werden). Dies ist speziell bei Test-Ausgaben wichtig zu wissen, da man sonst den Fehler an einer ganz falschen Stelle vermuten könnte.

Auch die Eingaben werden zwischengespeichert und erst dann an `scanf` übergeben, wenn die Zeile mit Enter (*Carriage Return*) abgeschlossen wird. Der erste Aufruf von `scanf` blockiert (wartet) so lange, bis der Benutzer eine Zeile eingegeben hat. Alle Eingaben werden dann zu einem Zeichenstrom zusammengefasst, aus dem sich `scanf` die gewünschten Zeichen herausfischt. Leerzeichen werden dabei meist ignoriert, bei nicht passenden Zeichen wird das Lesen abgebrochen. Es werden alle passenden Zeichen aus der Zeile verwendet, bis ein nicht passendes Zeichen kommt oder der Strom zu Ende ist. Die restlichen Zeichen bleiben für weitere `scanf` im Buffer. Erst wenn keine Zeichen mehr im Eingabebuffer übrig sind, wird ein weiterer `scanf`-Aufruf auf eine neue Benutzereingabe warten.

Was passiert also nun, wenn zwei Integerzahlen eingelesen werden sollen, der Benutzer aber zunächst eine Fließkommazahlen eingibt?

```
int a=2, b=2;
if(scanf("%d", &a)==0)
{
        printf ("Unzulaessige Eingabe!");
        return 1;
}
if(scanf("%d", &b)==0)
{
        printf ("Unzulaessige Eingabe!");
        return 1;
}
```

Der Benutzer gibt folgendes ein:

```
11.75
5
```

Man könnte nun vermuten, dass `a` den Wert 11 oder 12 und `b` den Wert 5 erhält. Die Realität sieht aber anders aus:

Nach dem Zeilenumbruch (Enter) beginnt das erste `scanf` mit der Verarbeitung der Eingabe. Es liest die Zahl 11 und bricht beim Punkt ab, da dieses Zeichen ja in einer Integerzahl nicht vorkommen kann. Die Variable `a` erhält also den Wert 11. Nun macht das zweite `scanf` weiter; es sind noch Zeichen im Eingabestrom vorhanden, also muss keine weitere Eingabe abgewartet werden. Es folgt aber immer noch der Punkt, das Einlesen wird also wieder abgebrochen und die Variable `b` bleibt unverändert.

5 Vergleichsoperatoren und logische Operatoren

5.1 Vergleichsoperatoren

In einem Programm ist es häufig notwendig, zwei Werte miteinander zu vergleichen. Zu diesem Zweck gibt es in C eine Reihe von Vergleichsoperatoren, z.B. den Operator „kleiner“, der mit dem Symbol < dargestellt wird. Für eine Wiederholung, was Operatoren, Operanden und Ausdrücke sind, lesen Sie bitte im Kapitel 3.1.4 nach.

Für den Vergleich zweier Werte gibt es die in Tabelle 5.1 gegebenen Vergleichsoperatoren. Man beachte, dass manche Operatoren aus zwei Zeichen bestehen. Machen Sie sich außerdem bitte bewusst, dass zwischen == (gleich) und = (Zuweisung) eine hohe Verwechslungsgefahr besteht, was zu unerwarteten Programmabläufen führen kann! Später werden Sie konkrete Fälle kennenlernen.

gleich	`==`
ungleich	`!=`
kleiner	`<`
größer	`>`
kleiner oder gleich	`<=`
größer oder gleich	`>=`

Tabelle 5.1: Vergleichsoperatoren

Mit den Vergleichsoperatoren kann man auch zwei Ausdrücke miteinander vergleichen. Ein korrekter Vergleichs-Ausdruck ist also z.B. auch:

```
a + b < c - d
```

wobei `a`, `b`, `c` und `d` beliebige Datentypen sein können.

Welches Ergebnis liefert also die folgende Vergleichsoperation?

```
1.9 > 2.0
```

Die Antwort eines Vergleiches lautet ja oder nein, bzw. in der Booleschen Algebra TRUE oder FALSE. In C existiert aber kein logischer Datentyp, es gibt keine logischen Variablen. Das Ergebnis eines Vergleiches ist hier

- 0 für nein (FALSE)
- 1 für ja (TRUE; eigentlich gelten alle Werte außer 0 als TRUE)

also ein Integer-Wert. Das Ergebnis kann auch in einer `int`-Variablen abgelegt werden. Der Vergleich `1.9 > 2.0` liefert also den Wert 0, mit `i = 1.9 > 2.0` hat also `i` den Wert 0.

Beispiele für Vergleichsoperationen sind also:

`17 == 120`	0 (FALSE)
`8.4 < 9.3`	1 (TRUE)
`9 != 10`	1 (TRUE)

5.2 Häufige Fehler

Wie bereits erwähnt, werden = und == gerne verwechselt. Der Compiler kann dies meist nicht bemerken und daher auch nicht warnen, da eine Zuweisung ja auch den Wert „zurückgibt“, der zugewiesen wurde: `a = 5` hat also den Wert 5, was dem Booleschen Wert TRUE entspricht! Hier wird also nicht der Inhalt von `a` mit 5 verglichen (Antwort: ja oder nein), sondern `a` wird (fälschlicherweise) auf 5 verändert.

Ein Vergleich `a < b < c` ist zwar möglich, bedeutet aber etwas anderes, als in der Mathematik: dieser Ausdruck wird von links nach rechts abgearbeitet. Zunächst wird also `a < b` verglichen, das Ergebnis ist 0 oder 1; dieses Ergebnis wird dann auf `< c` verglichen! Einige Compiler geben an dieser Stelle übrigens sogar eine Warnung aus.

Vergleiche auf Gleichheit (oder Ungleichheit) bei `float` oder `double` sind sehr problematisch. Diese Vergleiche sind natürlich möglich, führen aber häufig zu überraschenden Ergebnissen: da nicht jede Zahl exakt dargestellt werden kann, kommt es natürlich auch bei Berechnungen im Computer zu Rundungsfehlern. Der Vergleich `1.5 * 2 == 3` könnte also durchaus FALSE sein! Und das, obwohl bei der Ausgabe beide Werte identisch aussehen: bei der Ausgabe wird ja meist auf weniger Stellen gerundet. Die einzige Stelle, an der ein derartiger Vergleich häufig Sinn macht, ist, ob ein Operand ungleich `0.0` ist, bevor man ihn als Divisor verwendet.

5.3 Logische Operatoren

Obwohl es keinen logischen Datentyp in C gibt, können logische Verknüpfungen durchgeführt werden, z.B. die Ergebnisse von zwei Vergleichsoperationen `a > b` und `a < c`. Für diesen Zweck gibt es drei Operatoren, die den logischen Verknüpfungen Konjunktion, Disjunktion und Negation entsprechen. Sie sind in Tabelle 5.2 auf der nächsten Seite aufgelistet.

Wenn `a`, `b`, `c` und `d` Variablen von entsprechenden Typen sind, kann man also folgende logische Verknüpfungen bilden:

Logisches UND	`&&`
Logisches ODER	`||`
Logische Negation (NICHT)	`!`

Tabelle 5.2: Logische Operatoren. Man beachte, dass auch hier zwei Zeichen einen Operator bilden.

```
(a+b < c-d) || (a==b)
```

oder

```
(a > b) && (a < c)
```

Logisch verknüpft werden hier aber eigentlich die zwei Integer Werte 0 oder 1 für FALSE oder TRUE. Allgemein wird aber in C die logische Verknüpfung auf alle Werte erweitert, mit der Zuordnung:

- 0 entspricht FALSE (falsch, nein)
- Jeder andere Wert (ungleich 0) entspricht TRUE (wahr, ja)

Das Ergebnis einer solchen logischen (booleschen) Verknüpfung ist aber wiederum logisch, also hier 0 oder 1. Man kann damit auch mit einem logischen Wert arithmetisch weiterrechnen:

```
(a > b) + 1
```

denn das Vergleichsergebnis ist ein Integerwert, 0 oder 1, d.h. das Gesamtergebnis ist 1 oder 2.

```
#include <stdio.h>

int main()
{
  int i, j, k;
  i = (17 == 120);     /* 0 (FALSE), weil "17 gleich 120" nicht wahr ist */
  j = (8.4 < 9.3);     /* 1 (TRUE), weil "8.4 kleiner 9.3" wahr ist */
  k = (9 != 10);       /* 1 (TRUE), weil "9 ungleich 10" wahr ist */
  printf ("Werte: i = %d, j = %d, k = %d.\n", i, j, k);
  printf ("i || k (i ODER k) ist %d.\n", i || k);
  printf ("i && k (i UND k) ist %d.\n", i && k);
  printf ("!i && k ((NICHT i) UND k) ist %d.\n", !i && k);
  return 0;
}/* main */
```

Listing 5.1: Logische Operatoren

5.3.1 Logische Verknüpfungen

Mit logischen Operatoren werden logische Elemente nach den Regeln der Boolschen Algebra verknüpft. In C gibt es aber keinen logischen Datentyp, deshalb werden hier die bekannten arithmetischen Datentypen logisch verknüpft, nach der Regel: 0 entspricht FALSE (falsch, 0), ungleich 0 entspricht TRUE (wahr, 1). Das Ergebnis ist eigentlich vom Typ logisch, aber hier Integer mit den möglichen Werten 0 (FALSE) und 1 (TRUE).

Beispiel:

```
int a, b, c;
a = 3; b = 4;
```

Was ergibt die logische Verknüpfung `c = a && b`**?** Die UND-Verknüpfung ergibt TRUE, also den Integerwert 1, da beide Elemente ungleich 0 (also TRUE) sind.

Vergleichen wir nun damit die bitweise logischen Operatoren `&`, `|`, `^`.

Was ergibt die bitweise logische Verknüpfung `c = a & b`**?** Hier müssen wir das Bitmuster betrachten:

```
a: 3    00 ... 0011
b: 4    00 ... 0100
a & b:  00 ... 0000
```

Bit für Bit verknüpft ergibt das Bitmuster alles 0, also den Zahlenwert 0, was wiederum logisch betrachtet dem Wert FALSE entspricht.

Gleiches gilt für die Negation, bei der logischen Negation (`!`) wird aus dem Wert 0 eine 1, aus ungleich 0 eine 0. Bei der bitweisen Negation wird aus `00 ... 000` (das dem Zahlenwert 0 entspricht) lauter 1, also intern `11 ... 111`, was dem Zahlenwert -1 entspricht, bzw als unsigned dem maximalen Zahlenwert. In diesem Fall entspricht die bitweise der logischen Negation, was ansonsten keineswegs der Fall ist.

Was ergibt die bitweise Negation der Zelle `a` **mit dem Wert** `3`**?** Hier erhält man das Bitmuster `11 ... 1100` (nachdem 3 dem Bitmuster `00...0011` entspricht).

5.4 Priorität der Operatoren

Wie die arithmetischen Operatoren sind die Vergleichsoperatoren und die logischen Operatoren mit einer Priorität verbunden, welche die Reihenfolge der Abarbeitung festlegt. Wir können also unsere Prioritätenliste ergänzen. Die neue Liste zeigt Tabelle 5.3 auf der nächsten Seite.

Primary Operatoren	`()`
Unäre Operatoren	`- + ! ++ --`
Multiplikationsoperatoren	`* / %`
Additionsoperatoren	`+ -`
Shiftoperatoren	`<< >>`
Vergleichsoperatoren	`< <= > >=`
Gleichheitsoperatoren	`== !=`
Bitweises UND	`&`
Bitweises XOR	`^`
Bitweises ODER	`|`
Logisches UND	`&&`
Logisches ODER	`||`
Zuweisungsoperatoren	`=`

Tabelle 5.3: Operatorprioritäten

Die Gruppen sind nach fallender Priorität geordnet. Man beachte, dass hier in C die Vergleichsoperatoren und Gleichheitsoperatoren in der Rangordnung nochmals gestaffelt sind. Auch das UND ist höherwertig als das ODER. Alle diese Operatoren sind aber in der Rangordnung hinter den arithmetischen Operationen zu finden. Beachten Sie aber, dass der Negationsoperator `!` stärker bindet als UND und ODER, d.h. in der Reihenfolge der Ausführung vorgeht.

Die erste Prioritätsstufe nach den Klammern `()`, die auch als Primary Operatoren bezeichnet werden, haben die unären (unary) Operatoren (sie werden nur mit einem Operanden verwendet), d.h. die Negations-Operatoren `-` und `!` besitzen die höchste Priorität, die Negation wird also vor allen anderen Operatoren zuerst durchgeführt. In der Gruppe der Unary Operatoren sind auch das positive Vorzeichen +, aber auch die noch nicht eingeführten Zeigeroperatoren (siehe dazu die Kapitel 9.1 und 11.2.8) zu finden, sowie der Inkrement- und Dekrement-Operator (Kapitel 3.5.5).

Die logischen Verknüpfungen besitzen die niedrigste Priorität und werden deshalb zum Schluss nach den Vergleichs-Operatoren ausgeführt.

Die Reihenfolge der Ausführung von Operationen kann durch Einsatz von runden Klammern `()` verändert werden. Wie in der Mathematik werden auch hier die Ausdrücke innerhalb von Klammern zuerst berechnet.

Da die Multiplikations-Operatoren eine höhere Priorität als die Additions-Operatoren besitzen, wird bei der Berechnung eines Terms die aus der Mathematik bekannte *Punkt-vor-Strich-Regel* eingehalten. Als letztes in der Rangordnung wird die Zuweisung ausgeführt, die in C ja auch syntaktisch einen Operator darstellt (siehe auch Kapitel 3.5).

Man beachte, dass diese Prioritätenliste nur für C gilt, in anderen Programmiersprachen jedoch zum Teil unterschiedliche Prioritäten gelten.

Welche Klammern in diesen Beispielen waren also unnötig?

```
(a+b < c-d) || (a==b)
(a > b) && (a < c)
```

Alle Klammern waren unnötig. Sie sind aber dennoch sinnvoll, da sie den Code lesbarer machen und Fehler durch falsch vermutete Prioritäten vermeiden!

```
a+b < c-d || a==b
a > b && a < c
```

Arithmetische Operationen werden zuerst, dann Vergleichsoperationen und danach die Logischen Verknüpfungen ausgeführt.

Im Beispiel

```
(a > b) + 1
```

ist die Klammer aber erforderlich, da ansonsten die Rangordnung

```
a > b + 1
a > (b + 1)
```

ergeben würde, was natürlich ein anderes Ergebnis liefert.

Bemerkenswert ist auch, dass Boolesche Ausdrücke wie etwa

```
p || q
```

von links nach rechts ausgewertet werden, d.h., bereits wenn der erste Term `p` den Wert TRUE hat, bekommt der gesamte Ausdruck den Wert TRUE, ohne dass der zweite Term q ausgewertet wird. Das bringt in manchen Situationen Vorteile beim Programmieren.

Genauso ist es mit dem folgenden Ausdruck:

```
p && q
```

Wenn `p` den Wert FALSE hat, bekommt der ganze Ausdruck den Wert FALSE, ohne dass der zweite Term `q` ausgewertet wird.

Welchen Wert nimmt der Ausdruck an, wenn `p` den Wert TRUE hat? Wenn `p` immer TRUE ist, ist das Ergebnis immer gleich `q`.

- `p && q` bedeutet: Falls `p` den Wert TRUE hat, hat das Ergebnis den Wert von `q`, sonst FALSE.
- `p || q` bedeutet: Falls `p` den Wert TRUE hat, ist das Ergebnis TRUE, sonst der Wert von `q`.

Da in C eine automatische Typkonvertierung durchgeführt wird, ist das Ergebnis eines Ausdrucks mit unterschiedlichen Typen von dem Typ des höherwertigen Operanden. Das Ergebnis von Vergleichs-Operationen ist immer vom Typ `int`.

Sogar `char`-Variablen, also einzelne Zeichen, können aufgrund ihres jeweiligen ASCII-Wertes (vgl. Anhang B.1) mit den Vergleichs-Operatoren <, >, <=, >= und ==, != verglichen werden. Es kann sogar mit ihren ASCII-Werten gerechnet werden, hierfür ist keine Typkonvertierung notwendig, sie werden automatisch nach `int` konvertiert.

Dies gilt allerdings nur für Einzelzeichen! Für Zeichenketten (Strings) sind besondere Routinen erforderlich, auf die wir später noch eingehen werden.

6 Kontrollstrukturen

Wir haben bisher nur einfache Anweisungen kennengelernt, allgemein eine Wertzuweisung. Diese zählen zusammen mit dem Funktionsaufruf, auf den wir später eingehen, zu den sogenannten einfachen Anweisungen. Syntaktisch ist in C jeder Ausdruck zusammen mit einem Strichpunkt eine Anweisung. Daneben gibt es strukturierte Anweisungen, die ihrerseits aus (mehreren) anderen Anweisungen bestehen. Dazu zählen bedingte Anweisungen und Wiederholungsanweisungen.

Beginnen wir mit den bedingten Anweisungen.

6.1 Bedingte Anweisungen: `if` und `switch`

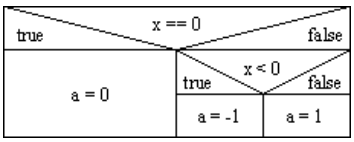
6.1.1 `if`-Anweisung

Eine Aufgabe kann es erforderlich machen, dass an einer bestimmten Stelle des Programms je nach Situation verschiedene Wege einzuschlagen sind. Dafür gibt es bedingte Anweisungen:

```
bedingteAnweisung ::= if-Anweisung | switch-Anweisung
```

Beispiel für die `if`-Anweisung:

```
if (x == 0)
  a = 0;
else
  if (x < 0)
    a = -1;
  else
    a = 1;
```



Welchen Wert erhält also die Variable `a`, wenn `x == -10` ist? Die Bedeutung der `if`-Anweisung ist gemäß dem normalen Sprachgebrauch offensichtlich: Ist der „logische Ausdruck“ in der Klammer nach `if` erfüllt (`TRUE`, also ungleich 0), wird die folgende Anweisung ausgeführt. Unser Beispiel lautet also in die Alltagssprache übersetzt: Wenn `x` gleich 0 ist, dann erhält `a` den Wert 0, wenn `x` kleiner 0 ist, dann erhält `a` den Wert -1, ansonsten bekommt die Variable `a` den Wert 1.

Wenn also `x` den Wert -10 hat, erhält `a` den Wert -1.

Seien Sie an dieser Stelle noch einmal auf die Verwechslungsgefahr von Zuweisungs- und Vergleichsoperator hingewiesen:

```
if (x = 0)
  a = 0;
```

ist syntaktisch richtig, hat aber keinen Sinn. Denn zunächst wird der Variablen `x` der Wert 0 zugewiesen und anschließend geprüft, ob dieser Ausdruck ungleich 0 ist. Der Ausdruck `x=0` hat natürlich immer den Wert 0, damit wird die Anweisung `a=0;` niemals ausgeführt.

Die Möglichkeit, eine Zuweisung in den Bedingungsteil einzubauen, ermöglicht es dem Programmierer, mehrere Schritte in einem zusammenzufassen, ist jedoch vor allem für den Anfänger eine große Gefahrenquelle. Das gleiche gilt auch für die Wiederholungsanweisungen (vgl. Kapitel 6.2).

Die `if`-Anweisung hat die Form

```
if ( Ausdruck )
  Anweisung1
else
  Anweisung2
```

Der alternative Pfad `else` kann auch entfallen:

```
if ( Ausdruck )
 Anweisung
```

Dann wird, wenn der Ausdruck `FALSE` (also 0) wird, nichts ausgeführt.

Der *Ausdruck* in den Klammern wird also berechnet; das Ergebnis ist damit die Bedingung für die Programmverzweigung. Die gesamte `if`-Konstruktion ist syntaktisch wiederum eine Anweisung, die in einem Programm stehen kann; an jeder Stelle, an der eine Anweisung erlaubt ist.

In unserem Beispiel von vorher

```
if (x==0)
  a = 0;
else
  if (x<0)
    a = -1;
  else
    a = 1;
```

haben wir bereits zwei `if`-Anweisungen verschachtelt. Im `else`-Pfad steht als *Anweisung2* erneut eine `if`-Anweisung.

Ein allgemeines Beispiel:

```
if ( Bedingung1 ) Anweisung1
  else if ( Bedingung2 ) Anweisung2
    else if ( Bedingung3 ) Anweisung3
      else Anweisung4
```

Dabei werden die Bedingungen 1 bis 3 der Reihe nach überprüft, und sobald eine den Wert `TRUE` aufweist, wird die zugehörige Anweisung ausgeführt. Es werden dann keine weiteren Bedingungen mehr getestet. Beachten Sie, dass sich jedes `else` auf das direkt vorhergehende `if` bezieht. Man sollte obigen Ausdruck also besser so formatieren:

```
if ( Bedingung1 )
  Anweisung1
  else
    if ( Bedingung2 )
      Anweisung2
    else
      if ( Bedingung3 )
        Anweisung3
      else
        Anweisung4
```

Welche Anweisung (Nummer) wird ausgeführt, wenn keine der drei Bedingungen erfüllt ist? Dann läuft das Programm jeweils in den `else`-Pfad und schließlich wird *Anweisung4* ausgeführt.

Normalerweise werden in einem Programm die Anweisungen sequentiell in der Reihenfolge abgearbeitet, wie sie niedergeschrieben wurden. Mit der `if`-Anweisung haben wir hier die erste Möglichkeit diesen Ablauf zu verändern, eine Programmverzweigung, abhängig von einer Bedingung, die im Programm zur Laufzeit berechnet wird.

Die Bedingung für die `if`-Verzweigung steht in einer runden Klammer. Die Bedingung kann ein einfacher Vergleich oder eine logische Verknüpfung mehrerer Bedingungen sein. Da es in C den logischen Datentyp eigentlich nicht gibt, ist hier die Bedingung ein allgemeiner Ausdruck, der nach den bekannten Regeln berechnet und dann, wie in Kapitel 5.3 besprochen, als „logische“ Bedingung verwendet wird:

- 0 entspricht falsch (nein, false)
- Jeder Wert ungleich 0 entspricht wahr (ja, true)

Der Wert des Ausdrucks kann dabei auch ein Gleitkommawert sein. Als Bedingung sind aber Integer-Ausdrücke zu empfehlen, da ein Gleitkomma-Ausdruck durch Rundungseffekte u.U. nie den exakten Wert `0.0` (falsch) annimmt.

```
#include <stdio.h>                /* nötig für Ein-/Ausgabe */

int main()                        /* Programm-Kopf */
{
  int x, a;

  printf ("Bitte geben Sie den Wert von x ein: ");
```

```
    if (scanf("%d", &x)==0)
    {
        printf   ("Unzulaessige Eingabe!");
        return 1;
    }

    if  (x == 0)
      a = 0;
    else
      if  (x < 0)
        a = -1;
      else
        a = 1;

    printf  ("Werte: x = %d, a = %d.\n", x,  a);
    return 0;
}/* main */
```

Listing 6.1: if-else
Das Programm belegt mittels `if-else` eine Variable abhängig von einer Benutzereingabe. Die zugehörige Datei finden Sie in Moodle unter den ergänzenden Dokumenten zum Skript unter Kapitel 6.

Blöcke In jedem der beiden Pfade steht jeweils nur eine Anweisung, damit wäre die `if`-Verzweigung sehr eingeschränkt. Anstelle einer einzelnen Anweisung kann aber auch eine Folge von Anweisungen stehen. Dazu muss der ganze Block von Anweisungen in geschweifte Klammern eingeschlossen werden.

```
{
  Anweisung1
  Anweisung2
  /* ... */
  AnweisungN
}
```

Ein solcher Block von Anweisungen ist syntaktisch eine Anweisung. Eine solche Block-Anweisung kann also in jedem Pfad der `if`-Verzweigung stehen, es können damit in jedem Fall nicht nur eine Anweisung, sondern eine ganze Reihe von Anweisungen ausgeführt werden.

- Es wird empfohlen, immer geschweifte Klammern zu verwenden, auch wenn es sich nur um eine einzelne Anweisung in der if -Verzweigung handelt. Das Programm wird damit besser lesbar; Fehler durch nachträglich eingefügte Anweisungen werden vermieden.

- Hinter der geschlossenen Klammer darf kein Strichpunkt stehen, da sonst ein nachfolgendes `else` der vorangehenden `if`-Anweisung nicht mehr zugeordnet werden kann (der Strichpunkt ist eine leere Anweisung).

Welchen Inhalt haben die Zellen a, b und k nach dieser Anweisung?

```
int k=0, a=123, b=456;

if (k < 0){           /* -                  */
  a = 0;     /* | Block-Anweisung  */
  b++;       /* |                  */
}
else {       /* -                  */
  a = 1;     /* | Block-Anweisung  */
  b--;       /* |                  */
}            /* -                  */
```

Die Variable `k` ist (bleibt) gleich 0, es wird also der `else`-Pfad benutzt. Damit gilt am Ende `a==1` und `b==455`.

6.1.2 `switch`-Anweisung

Mehrfachverzweigungen (auch Tabellenentscheidungen genannt) kann man besser und übersichtlicher mit der `switch`-Anweisung erreichen. Ein Beispiel soll die Anwendung demonstrieren:

```
switch (monat){
  case  1: printf("Januar");    break;
  case  2: printf("Februar");   break;
  case  3: printf("März");      break;
  ...
  case 11: printf("November");  break;
  case 12: printf("Dezember");  break;
  default: printf("Fehler !");  break;
}
```

Wie an dem Beispiel unschwer zu erkennen ist, findet hier eine Programmverzweigung abhängig von der Variablen `monat` statt. Nimmt `monat` den Wert 3 an, so wird zu der Marke `case 3:` verzweigt und die Abarbeitung bei der dort stehenden Anweisungen fortgesetzt. Hier können beliebig viele Anweisungen folgen. In unserem Beispiel folgt die Anweisung `printf("März")`.

Die weiter folgenden Anweisungen sollen aber nicht mehr verwendet werden. Daher wurde hier die Anweisung `break` eingefügt. `break` beendet die `switch` -Anweisung.

Andernfalls wïrden alle weiteren `printf`-Anweisungen auch noch ausgeführt werden. Nimmt `monat` einen Wert an, für den keiner der angegebenen Fälle zutrifft, so wird die nach der Marke `default:` stehende Anweisung `printf("Fehler !")` ausgeführt. `default` darf auch weggelassen werden. In diesem Fall würde dann gar nichts ausgeführt.

Etwas modifiziert kann unser (etwas unsinniges) Beispiel auch wie folgt aussehen:

```
switch (monat){
  case  8: printf("Sommerferien");
           break;
  case  9:
  case  3:
  case  4:
  case 10: printf("Prüfungen!");
           break;

  case 12: tage = tage + 31;
           break;

  case  5:
  case  6:
  case  7: printf("Sommersemester");
           break;

  case  2: if(schaltj) t = 29;
           else t = 28;
           tage = tage + t;

  case  1: tage = tage + 31;
           break;

  default: printf("Fehler!");
           break;
}
```

Was wird hier für den Fall `monat==0` oder `monat==2` ausgeführt? Für `monat==0` ist keine eigene `case`-Marke angegeben, also wird die `default`-Anweisung ausgeführt. Ist `monat==2`, wird die `if(schaltj)`-Anweisung ausgeführt. Dann wird die Ausführung bis zur nächsten `break`-Anweisung fortgesetzt (die Marken selbst stören dabei nicht). Insgesamt werden dann also (wenn `monat==2`) folgende Anweisungen ausgeführt:

```
if (schaltj) t = 29;
else t = 28;
```

```
tage = tage + t;
tage = tage + 31;
```

Für die Fälle 5, 6 und 7 wird nur

```
printf("Sommersemester");
```

abgearbeitet und dann die `switch`-Anweisung verlassen. Wie in dem Beispiel zu sehen, ist die Reihenfolge der `case`-Fälle beliebig. Eine fehlende `break`-Anweisung kann durchaus beabsichtigt und sinnvoll sein, wie im Beispiel bei `case 5:`, `case 6:` und `case 2:`.

Die allgemeine Form der `switch`-Anweisung lautet:

```
switch ( Ausdruck )
{
  case Fall : Anweisungen     break;
  ...
  default: Anweisungen
}
```

Der Ausdruck, der die Verzweigung steuert, sollte ganzzahlig sein, nur damit ist eine sinnvolle Verzweigung auf einen Fall möglich. `Fall` ist ebenfalls eine ganzzahlige Konstante oder Konstantenausdruck. Hat `Ausdruck` den Wert dieser Konstanten `Fall`, so beginnt die Abarbeitung nach der Marke `case Fall:` bis zum nächsten `break`.

Trifft keiner der angegebenen Fälle zu, so erfolgt ein Sprung zur Marke `default:`, sofern vorhanden, ansonsten wird keine der Anweisungen ausgeführt. `Fall` ist eine ganzzahlige Konstante, kann damit auch ein einzelnes Zeichen sein (z.B. `'A'`), aber keine Zeichenkette (nicht `"ABC"`).

Die `switch`-Anweisung ist eine Verallgemeinerung der `if`-Anweisung, die gleichzeitig mehr als zwei Fälle zulässt. Eine sinnvolle Verwendung findet die `switch`-Anweisung immer dann, wenn eine Tabellenentscheidung zu treffen ist, wenn also mehr als zwei alternative Fälle zu betrachten sind.

```
#include <stdio.h>                /* nötig für Ein-/Ausgabe */

int main()                        /* Programm-Kopf */
{
  int monat;

  printf ("Bitte  geben Sie die Nummer des Monats ein: ");
  if (scanf("%d", &monat)==0)
  {
      printf  ("Unzulaessige Eingabe!");
```

```
        return 1;
    }
    printf ("Es handelt sich um den Monat ");

    switch (monat) {
    case 1: printf ("Januar");      break;
    case 2: printf ("Februar");     break;
    case 3: printf ("März");        break;
    case 4: printf ("April ");      break;
    case 5: printf ("Mai");         break;
    case 6: printf ("I ");
    case 7: printf ("I vor ");
    case 8: printf ("August");      break;
    case 9: printf ("September");   break;
    case 10: printf ("Oktober");    break;
    case 11: printf ("November");   break;
    case 12: printf ("Dezember");   break;
    default: printf ("Fehler! ");   break;
    }

    printf ("!\ n");
    return 0;
}/* main */
```

Listing 6.2: switch

Beachten Sie insbesondere die Auswirkungen der fehlenden `breaks` für den Fall, dass der Benutzer 6 oder 7 eingibt.

6.2 Wiederholungsanweisungen: `while`, `do ... while` und `for`-Schleifen

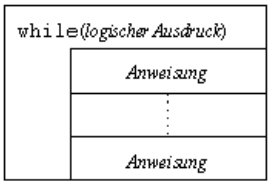
6.2.1 `while`-Schleifen

Es kommt häufig vor, dass eine Anweisung oder eine Folge von Anweisungen so oft wiederholt werden soll, solange eine bestimmte Bedingung erfüllt ist (iterative Schleife). Dies lässt sich mit der `while`-Anweisung erreichen:

```
while-Anweisung ::= " while " " ( " Ausdruck " ) " Anweisung .
```

Ein einfaches Beispiel:

```
int i = 0;
while (i < 10)
  i = i + 3;
printf("%d\n", i );
```



Bei der `while`-Anweisung wird die Bedingung (log. Ausdruck) zu Beginn eines jeden neuen Durchlaufs geprüft. Falls der Ausdruck also von Anfang an `FALSE` ist, wird die Anweisung gar nicht ausgeführt. Auch hier kann natürlich anstelle einer Anweisung eine Reihe von Anweisungen in geschweiften Klammern stehen, eine Block-Anweisung.

Achten sie bei Wiederholungsanweisungen darauf, dass auf den Abbruch der Schleife hingearbeitet wird. Wenn die Bedingung unverändert erfüllt bleibt, entsteht eine unendliche Schleife, das Programm muss dann zwangsweise abgebrochen werden.

Die Wiederholung von identischen Ausdrücken innerhalb der Schleife und der Schleifenbedingung sollte man vermeiden, da sie ansonsten in jedem Durchlauf identisch berechnet werden müssen. Mit identischen Ausdrücken sind hier Ausdrücke gemeint, die bei jedem Durchlauf der Schleife das selbe Ergebnis liefern. Sie würden dann jedesmal neu berechnet, was natärlich unsinnig ist. Man sollte solche Berechnungen vor der Schleife durchführen. übrigens erkennen dies manche Compiler (in der Optimierungsstufe) und nehmen solche Ausrücke selbst aus der Schleife heraus.

6.2.2 `do-while`-Schleifen

Eine weitere Wiederholungsanweisung ist die `do-while`-Anweisung.

```
do-while-Anweisung ::= "do" Anweisung "while" "("Ausdruck")" ";" .
```

Der wesentliche Unterschied zur `while`-Anweisung besteht darin, dass die Bedingung jedesmal nach der Ausführung der Anweisungen geprüft wird, d.h. die Anweisungsfolge wird immer mindestens einmal durchlaufen.

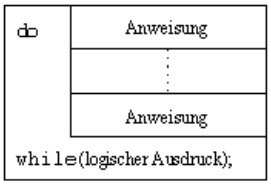
Bei der `do-while`-Anweisung wird die Schleife solange durchlaufen, bis der Ausdruck nach `while`, also die Schleifenbedingung, den Wert 0 (`FALSE`) annimmt.

Die Prüfung der Bedingung erfolgt also nach dem Ende der Anweisungen. Als „logischer Ausdruck“ gelten die bekannten Zuordnungen, also:

- 0 entspricht `FALSE` (nein)
- ungleich 0 (meist 1) entspricht `TRUE` (ja)

Ein einfaches Beispiel:

```
int i = 0;
do
  i = i + 3;
while (i < 10);
printf("%d\n", i);
```



Dieses Beispiel wird mit den angegebenen Werten genau das Gleiche ausgeben wie das Beispiel für die `while`-Schleife. Wenn als Startwert aber statt 0 z.B. 20 genommen wird, dann wird die `while`-Schleife gar nicht ausgeführt, während die `do-while`-Schleife einmal durchläuft.

Auch bei der `do-while`-Anweisung muss man natürlich darauf achten, dass auf die Abbruchbedingung hingearbeitet wird.

Neben der `while`- und der `do-while`-Anweisung gibt es noch eine weitere Anweisungen dieser Art, nämlich die `for`-Anweisung, zu der wir in in Kapitel 6.2.4 kommen.

6.2.3 `break`-Anweisung

Wenn die Schleifenbedingung konstant auf `TRUE` (ungleich 0) bleibt, hat man eine Endlosschleife programmiert. Dies kann sehr wohl sinnvoll und beabsichtigt sein. Hier muss man auf jeden Fall eine Abbruchbedingung in der Schleife vorsehen, die das Verlassen der Schleife an einem beliebigen Punkt der Anweisungsfolge zulässt. Der Abbruch der Schleife an einer beliebigen Stelle wird durch die Anweisung `break;` erreicht.

Beispiel für eine Endlosschleife:

```
int n=4, i=0, x=1, y;
while (1){
  i = i + 1;
  if (i > n) break;
  x = x * i;
}
y = x;
```

Wenn `i>n` ist, dann wird die `break`-Anweisung ausgeführt, d.h. die `while`-Schleife – hier sonst eine Endlosschleife – wird sofort verlassen, und das Programm macht bei der Anweisung `y=x` weiter.

Die `break`-Anweisung kann natürlich in jeder `while`- oder `do-while`-Schleife verwendet werden, um die Schleife unter einer gegebenen Bedingung vorzeitig zu verlassen. Sie macht Schleifen aber häufig unübersichtlich: man sollte sich schon überlegen, ob es nicht einen besseren Aufbau für die Schleife gibt, und man auch ohne `break` auskommt.

6.2.4 `for`-Schleifen

Die `for`-Anweisung ist eine Wiederholungsanweisung mit folgender Form:

```
for ( Ausdruck1 ; Ausdruck2 ; Ausdruck3 )
  Anweisung
```

In der EBNF-Schreibweise dargestellt:

```
for-Anweisung ::= "for" "("Ausdruck";"Ausdruck";"Ausdruck")" Anweisung .
```

Die einzelnen Ausdrücke sind:

Ausdruck1 ist der Initialisierungsausdruck, z.B. `i=0`
: Wird vor dem Eintritt in die Schleife ausgeführt, in der Regel eine Zuweisung.

Ausdruck2 ist die Schleifenbedingung, z.B. `i<n`
: Solange die Bedingung erfüllt (`TRUE`) ist, wird die Schleife durchlaufen.

Ausdruck3 die Weiterschaltung, z.B. `i++`
: Wird nach der Schleifen-Anweisung ausgeführt, meist ein Inkrement oder Dekrement

Man kann eine `for`-Schleife auch mit Hilfe einer `while`-Schleife schreiben. Nehmen wir folgendes kleine Beispiel:

```
int x, y;
for (x=0; x<10; x++){
  y = x * 2;
  printf("%i * 2 = %i\n", x, y);
}
```

Diese Schleife ist identisch zu folgender `while`-Schleife:

```
int x, y;
x = 0;
while (x < 10){
  y = x * 2;
  printf("%i * 2 = %i\n", x, y);
  x++;
}
```

Daraus ist zu erkennen, dass der Initialisierungsausdruck (Ausdruck1: `x=0`) vor der eigentlichen Schleife genau einmal ausgeführt wird. Die Schleifenbedingung (Ausdruck2: `x<10`) wird vor jedem Durchlauf der Schleife geprüft. Es kann also durchaus sein, dass die Schleifen-Anweisung gar nicht ausgeführt wird. Nach dieser Anweisung erst wird die Weiterschaltung (Ausdruck3: `x++`) ausgeführt; im ersten Durchlauf wird also lediglich die Initialisierung durchgeführt und die Bedingung geprüft.

Die `for`-Schleife wird meist dann verwendet, wenn eine feste Anzahl von Schleifendurchläufen erforderlich ist, oder ein Variable in konstanten Abständen erhöht oder verringert werden soll. Man kann aber statt der drei Ausdrücke beliebige andere Ausdrücke verwenden; meist führt das aber wieder nur zu unverständlichen Programmen.

Der Quellcodeabdruck 6.3 auf der nächsten Seite fasst nochmals die unterschiedlichen Schleifentypen in einem Beispiel zusammen. Mehr zu Schleifen finden Sie noch in Kapitel 8.1.2.

```
#include <stdio.h>          /* nötig für Ein-/Ausgabe */

int main()                  /* Programm-Kopf */
{
  int i=0, j, x, y, n=4;

  while (i < 10){           /* while-Schleife */
    i = i + 3;
  }
  printf ("while: %d \n", i );

  i = 0;
  do {                      /* do-while-Schleife */
    i = i + 3;
  } while (i < 10);
  printf ("do-while: %d\n", i );

  i=0;
  while (1){                /* Endlosschleife basierend auf while */
    i = i + 1;
    if (i > n) break;       /* Verlassen der Schleife, wenn "i>n" */
      x = x * i;
  }
  printf ("break: %d\n", i );

  for (x=0; x<10; x++){     /* for-Schleife */
    y = x * 2;
    printf ("%i * 2 = %i\n", x, y);
  }
  return 0;
}/* main */
```

Listing 6.3: Schleifen

7 Programmierstil (Beispiel)

Wir haben bereits in Kapitel 2.2.2 auf die wesentlichen Punkte hingewiesen (lesen Sie sich dieses Kapitel gerne noch mal durch). Nachdem Sie schon wesentlich mehr über C wissen, soll an dieser Stelle ein extremes Beispiel für ein syntaktisch richtiges und lauffähiges, aber formal schrecklich gestaltetes Programm vorgestellt werden. Überlegen Sie mal: Was macht das Programm 7.1?

```
#include <stdio.h>

int  main

(){ int  a=-2,b,c;printf(

"Fahrenheit/Celsius Temperaturtabelle"

); printf   (

"\n\n"

); for(a=-2;a<13;a++){b=

10*a;c=32+b*(9

/5); printf  ("  C= %d F= %d",

b,c); if (b==0)printf("  Gefrierpunkt von Wasser");if  (b== 100)printf

("  Siedepunkt von Wasser");printf("\n"); return 0;}}
```

Listing 7.1: Schlechtes Beispiel

Um derart unleserlichen Code zu vermeiden, sollten folgende Punkte beachtet werden:

- Jedes Programm sollte mit einem „Kopf“ versehen werden, der Auskunft über das Programm gibt. Wichtig ist vor allem, welche Funktion das Programm hat. Sinnvoll sind auch Erstellungsdatum und Verfasser.
- An den wichtigen Programmierschritten sollten Kommentare eingefügt werden; dies können durchaus noch mehr sein, als im Beispiel.

- Logisch zusammenhängende Teile sollten unbedingt durch Einrücken und Leerzeilen hervorgehoben werden.
- Verwendung von selbsterklärenden Variablen- und Funktionsnamen.

Versuchen Sie, diese Punkte bei der Programmierung immer zu beachten! Besonders bei längeren Programmen ist dies wichtig. Nur so können Ihnen auch mal andere helfen, Fehler in Ihrem Programm zu finden!

Wenn Sie das obige Programm compilieren und starten, dann ergibt sich genau die selbe Ausgabe wie bei folgendem:

```
#include <stdio.h>

int main()
{
  int zaehler;       /* Schleifenzähler */
  int gradCelsius;   /* Temperatur in Grad Celsius */
  int fahrenheit;    /* Temperatur in Grad Fahrenheit */

  printf ("Fahrenheit/Celsius Temperaturtabelle\n\n");

  for (zaehler=-2; zaehler<=12; zaehler++){
    gradCelsius = 10 * zaehler;   /* Celsius von -20 bis 120 */
    fahrenheit  = 32 + gradCelsius * (9/5);    /* Umrechnung */

    printf ("   C = %d", gradCelsius);
    printf ("   F = %d", fahrenheit);

    if (gradCelsius == 0){
      printf ("   Gefrierpunkt von Wasser");
    }

    if (gradCelsius == 100){
      printf ("   Siedepunkt von Wasser");
    }

    printf ("\n");

  }  /* Ende "for zaehler" */
  return 0;
}  /* Ende Hauptprogramm */
```

Listing 7.2: Gutes Beispiel

Zweifelsohne kann die zweite Version des Programms leichter verstanden werden als

die erste. So können Änderungen leicht vorgenommen und vorhandene Fehler gefunden werden.

Hat sich in obigem Beispiel eigentlich ein Fehler eingeschlichen? Ja, tatsächlich! Die Berechnung wird mit Integer-Werten durchgeführt. Die `gradCelsius`-Variable kann durchaus vom Typ `int` bleiben, aber `fahrenheit` sollten schon `float` sein (Rundung!). Wirklich falsch ist aber die Umrechnung: der Ausdruck `9/5` wird zu 1, und nicht 1.8 wie beabsichtigt.

8 Zusammengesetzte Datentypen

8.1 Datenfelder (Arrays)

8.1.1 Datentyp Array

Aus den einfachen Datentypen, die wir bisher kennengelernt haben, lassen sich auch kompliziertere Datentypen konstruieren. Einer dieser sogenannten strukturierten Datentypen ist der Datentyp *Array*. Arrays (Felder) bestehen aus einer festen Anzahl gleichartiger Elemente. Die einzelnen Elemente werden durch Indices gekennzeichnet.

Die Vereinbarung eines Arrays sieht beispielsweise so aus:

```
int a[N];
```

Arrays oder Felder bestehen aus einer festen Anzahl gleichartiger Elemente. Mit obiger Vereinbarung hat das Feld `a` insgesamt N Elemente, die alle vom Typ `int` sind. Die Indices laufen immer von 0 bis N-1. Das erste Element des Feldes hat also immer den Index 0, und der Index des letzten Elements ist immer um eins weniger als die bei der Vereinbarung angegebene Anzahl an Elementen!

Bei der Vereinbarung des Feldes muss `N` eine konstante ganze Zahl sein. Felder mit variabler Größe können nicht erzeugt werden.

Die Werte der einzelnen Elemente sind am Anfang nicht automatisch auf 0 gesetzt, sondern undefiniert, d.h. die Werte sind beliebig. Jedes Element des Feldes ist zu benutzen wie eine normale Variable, die ja auch zunächst initialisiert werden muss.

Das i-te Element des Feldes `a` wird durch `a[i]` dargestellt. Alle Elemente eines Feldes haben den gleichen Datentyp.

Beispiel:

```
char name[20];
```

Diese Variablendefinition bildet den aus anderen Programmiersprachen bekannten Typ `string` nach. C kennt diesen Typ nicht, kann aber trotzdem mit Zeichenketten als Feld von Einzelzeichen arbeiten. Näheres zu Zeichenketten finden Sie in Kapitel 8.2.1.

Die Elemente des Arrays werden mit `name[i]` dargestellt. Das 6. Element lautet also `name[5]`, das letzte Element ist `name[19]`. Wenn wir nun z.B. dem 6. Element unseres Feldes den Wert `'s'` zuweisen wollen, müssen wir schreiben:

```
name[5] = 's';
```

Die Elemente eines Feldes können wie normale Variable verwendet werden. Um das 6. Zeichen auszugeben, kann man folgendes schreiben:

```
printf(" %c", name[5]);
```

Beim Datentyp Array heißen die einzelnen Elemente Arrayelemente oder indizierte Variable.

Allgemein:

```
ArrayElement ::= Arrayvariable "[" Ausdruck "]" { "[" Ausdruck "]" } .
```

Beim Zugriff auf ein Arrayelement darf der Index also auch ein Ausdruck sein. Beispiele:

```
name[n-3]
a[5*i]
a[2*n-1]
```

Achtung: In C findet beim Zugriff auf Arrayelemente keine Indexprüfung statt! Wenn z.B. ein Feld `int f[20];` vereinbart wurde, gibt es keine Fehlermeldung oder Warnung, wenn man versucht, etwa auf das Element `f[24]` zuzugreifen, obwohl damit fremder, möglicherweise noch anderweitig benötigter Speicherplatz überschrieben wird. Genau dieses Überschreiben von irgendwelchen Speicherzellen kann aber verheerende Folgenden für den weiteren Programmablauf haben - es wird dann z.T. sehr schwierig, die Ursache zu finden.

Wenn das Programm an dieser Stelle abstürzt, hat man noch Glück. Oft ist es jedoch so, dass es erst später zu völlig unerwarteten Abläufen kommt und somit der Fehler schwer herauszufinden ist. Achten Sie deshalb bitte stets darauf, dass Sie in Ihren Programmen die Indexgrenzen von Feldern nicht überschreiten.

8.1.2 Schleifen und Arrays

Nehmen wir nochmals ein Beispiel:

```
int a[100], n=100;
```

Wenn wir nun sämtlichen Elementen dieses Arrays den Wert 0 zuweisen wollen, so könnte dies z.B. mit einer `while`-Schleife (siehe Kapitel 6.2.1) folgendermaßen geschehen:

```
int i=0;
while (i<n)
  { a[i]=0; i++; }
```

Einfacher geht es mit der `for`-Anweisung (vgl. auch Kapitel 6.2.4):

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

Ausdruck1 (`i=0`) und *Ausdruck2* (`i<n`) bilden die Grenzen, zwischen denen sich die Laufvariable bewegt. Die Schrittweite, mit der sich die Laufvariable bewegen soll, kann im *Ausdruck3* (`i++`) gewählt werden. Die Schleife wird also solange durchlaufen, bis die Laufvariable die in *Ausdruck2* gegebene Bedingung nicht mehr erfüllt, bzw. den dort gegebenen Grenzwert erreicht hat. Ist bei positivem Inkrement der Endwert kleiner als der Anfangswert, wird die Schleife überhaupt nicht durchlaufen. Auch hier ist darauf zu achten, dass keine unbeabsichtigte Endlosschleife entsteht.

8.1.3 Arrays kopieren

Wenn wir die beiden Arrays `name1` und `name2` gemäß

```
char name1[n], name2[n];
```

vereinbart haben, und dem Array `name2` dieselben Werte wie `name1` zuweisen wollen, so müssen wir Element für Element einzeln übertragen, z.B.:

```
for (i=0; i<n; i++)
  name2[i] = name1[i];
```

Es geht nicht einfach:

```
name2 = name1;
```

Man muss jedes ArrayElement einzeln zuordnen! Ganze Felder können nicht zugewiesen oder mit irgendeinem Operator verknüpft werden. Operationen wie elementenweise Addition oder Multiplikation sind nicht vorgesehen, sondern müssen selbst programmiert werden.

```
#include <stdio.h>      /* nötig für Ein-/Ausgabe */

int main()              /* Programm-Kopf */
{
  int i, a[50], b[50]; /*Deklaration von zwei int-Arrays mit 50 Elementen*/

  for(i=0; i<50; i++)  /*i durchläuft die Werte 0 bis 49*/
  {
    a[i] = i * i;      /*element a[i] wird mit i^2 belegt*/
  }
  i--;                 /*i von 50 auf 49 setzen*/
```

```
  while(a[i] > 1600)   /* alle  a[i] > 1600 (d.h.a[41]-a[49],
                          da 40*40 = 1600) werden gleich 0 gesetzt*/
  {
    a[i] = 0;
    i--;
  }

  for(i=0; i<50; i++) /* for  Schleife  um Feld a in Feld b zu kopieren*/
  {
    b[i] = a[i];
    printf ("b[%i]=%i\n", i, b[i]);   /* Ausgabe von b*/
  }
  return 0;
}/* main */
```

Listing 8.1: Arrays
Im Programm wird das Feld `a` zunächst initialisiert (1. `for`-Schleife), manipuliert (`while`-Schleife), dann kopiert (2. `for`-Schleife).

8.1.4 Mehrdimensionale Arrays

Es können auch mehrdimensionale Arrays erzeugt werden:

```
int matrix[100][50];      /* 2-dimensional */
int b[50][100][10];       /* 3-dimensional */
int i, j, k, l, m;
```

Auf die einzelnen Elemente kann man mittels der Elementvariablen (ArrayElement) zugreifen:

```
matrix[i][j]
b[k][l][m]
```

Das Array `matrix` enthält in unserem Beispiel $100 \cdot 50$, also 5000 Elemente von Typ `int`.

```
#include <stdio.h>      /* nötig  für  Ein-/Ausgabe */

int  main()             /* Programm-Kopf */
{
  int i, j;
  int  a[5][10];   /* Deklaration eines zweidimensionalen int-Arrays a*/

  for(i=0; i<5; i++)     /* i  durchläuft  die  Werte 0 bis 4 */
    for(j=0; j<10; j++) /* j  durchläuft  die  Werte 0 bis 9 */
    {
```

```
      a[i][j] = (i+1) * (j+1); /* element a[i][j] wird mit (i+1)*(j+1) belegt */
    }

  printf ("Das zweidimensionale Array a enthaelt das \"kleine Einmaleins\"!\n");
  for(i=0; i <5; i++) /*verschachtelte Schleife zur Ausgabe des Arrays*/
  {
    printf ("\n%der Einmaleins:\n",i+1);
    for(j=0; j<10; j++)
    {
      printf ("%2d\t",a[i][j]);
    } /* for j */
  }/*for i*/

  printf ("\n");
  return 0;
}/* main */
```

Listing 8.2: Mehrdimensionale Arrays
Das Programm zeigt, wie zweidimensionale Arrays/Felder zu verwenden sind.

8.2 Datentyp `char` und Zeichenketten (Strings)

8.2.1 Datentyp `char`

Kommen wir nun zu einem weiteren elementaren Datentyp, dem Typ `char`(von Englisch character = Zeichen). Die Variablen vom Typ `char` können als Werte alle zur Verfügung stehenden Zeichen annehmen. Diese Zeichen werden zwischen Apostrophe `'`, nicht zwischen Anführungszeichen `"` gesetzt. Konstanten dieses Typs sind also z.B.:

```
'a' '4' '-' '*' 'S'
```

Alle verfügbaren Zeichen bilden eine geordnete Menge, jedem Zeichen ist eine bestimmte Ordnungszahl zugewiesen, die vom verwendeten Code abhängt. Der am meisten verwendete Code ist dabei der ASCII-Zeichensatz (American Standard Code for Information Interchange). Im Anhang B.1 ist die ASCII-Code-Tabelle aufgeführt. Damit kann man die Ordnungszahl jedes Zeichens bestimmen.

Als Beispiel greifen wir das Zeichen `'2'` heraus. Sehen Sie sich die Tabelle im Anhang an und geben Sie dann die dezimale Ordnungszahl der Ziffer 2 an.

Die Ziffer wird (dezimal) mit 50 kodiert. Man kann die Ordnungszahl eines Zeichens auch in oktaler (mit führender 0: hier 062) oder hexadezimaler Form (mit vorangestelltem `0x`: hier `0x32`) angeben.

Wenn Sie sich eben die ASCII-Code-Tabelle angesehen haben, sind Ihnen wahrscheinlich die ersten Zeichen aufgefallen. Es handelt sich dabei um nicht darstellbare Zeichen (Kontrollzeichen), die verschiedene Funktionen haben. Das Symbol für newline,

'\n' kennen Sie bereits. Es wird als ein einzelnes Zeichen interpretiert; der Backslash \ gibt nur an, dass es sich um ein Kontrollzeichen handelt, welches mit der Tastatur nicht direkt dargestellt werden kann.

Welche Ordnungszahl (oktal) hat das Zeichen ff (form feed)? 014.

Zeichen, auch die nichtdarstellbaren Kontrollzeichen, werden in einer Zelle vom Typ `char` abgespeichert. Eine `char`-Variable hat eigentlich immer die Größe 1 Byte (8 Bit). Die Kodierung für ein Zeichen ist die o.g. Ordnungszahl. Da man in C diese `char`-Zelle wie Daten vom Typ `int` verwenden kann, kann man den Inhalt als Integerwert ausgeben oder auch damit rechnen. Der Wert entspricht also dieser Ordnungszahl.

Beispiel:

```
char ch;
ch = 'A';
printf("%d", ch);
```

erzeugt die Ausgabe `65` (oktal 101). Beachten Sie hier auch die Formatangabe `%d`, mit der Angabe `%c` wird das Zeichen selbst ausgeben, wie in folgendem Beispiel

```
printf("%c",ch);
```

erzeugt die Ausgabe `A`. Hier wird also das abgespeicherte Bitmuster unterschiedlich interpretiert, entsprechend der Formatangabe.

Wenn wir also die Variablenvereinbarung und Zuweisung haben:

```
char ch;
ch = 'g';
printf("%c\n",ch );
ch = ch + 10;
printf("%c\n",ch );
```

Welche Zeichen werden dann ausgegeben? Zunächst wird das `g` wieder ausgegeben. Zählt man in der ASCII-Tabelle um zehn Schritte weiter, so kommt man zum `q`; dieses Zeichen wird dann auch ausgegeben.

Wegen der geordneten Reihenfolge kann man Daten des Typs `char` auch miteinander vergleichen.

Welches Ergebnis liefert beispielsweise die folgende Operation?

```
'a' > 'A'
```

Variablen des Typs `char` kann man mit sämtlichen Vergleichsoperatoren, die wir bisher kennengelernt haben, vergleichen. Dasjenige Zeichen wird als „kleiner“ angesehen, dessen interne Darstellung (Ordnungszahl) kleiner ist. Das kleine „a“ ist also damit „größer“ als das große „A“ und obiger Ausdruck ist TRUE!

Es gilt z.B. auch:

```
'a' >  'b'    ist FALSE
'7' >= '0'    ist TRUE
'a' >  'A'    ist TRUE
```

Welchen Wert liefert die folgende Vergleichsoperation?

```
'\"' <= '-'
```

Auch dieser Vergleich ist TRUE.

Alle diese Betrachtungen gelten aber nur für Einzelzeichen. Völlig getrennt betrachten muss man Zeichenketten, also Strings (vgl. auch Kapitel 2.3.4).

```
#include <stdio.h>              /* nötig für Ein-/Ausgabe */

int main()                      /* Programm-Kopf */
{
  char klein = 'a', gross = 'A';
  int i;

  for(i = 0; i < 26; i++)
    printf ("%c", klein + i);
  printf ("\n");

  for(i = 0; i < 26; i++)
    printf ("%c", gross + i);
  printf ("\n");

  printf ("Abstand: %i\n", klein - gross);
  return 0;
}/* main */
```

Listing 8.3: char
Das Programm gibt das Alphabet in Klein- und Großbuchstaben, sowie den Abstand zwischen „a“ und „A“ im ASCII-Zeichensatz aus.

8.2.2 Zeichenketten (Strings)

Eine Zeichenkette, ein Text, besteht aus einer Folge von einzelnen Zeichen. Eine solche Zeichenkette haben wir bereits in unserem ersten Beispiel kennengelernt:

```
printf("Hurra, hurra ! ");
printf("\n");
printf("Asterix ist da ! \n");
```

Der auszugebende Text ist eine solche Zeichenkette, ein String:

```
"Hurra, hurra ! "
```

oder

```
"Asterix ist da ! \n"
```

Der String ist eine Folge von beliebigen Zeichen, er kann auch Sonderzeichen enthalten. Zeichenketten werden zwischen Anführungszeichen (") und nicht zwischen Apostrophe (') gesetzt.

Welches Zeichen ist in dem Beispiel ein Sonderzeichen? Wie ist es kodiert? Das Zeichen '\n' ist in dem String enthalten. Dieses Zeichen - als newline oder linefeed bezeichnet - ist in der ASCII-Tabelle mit der Kodierung (Ordnungszahl) 10 (oder oktal: 012) enthalten.

Die hier verwendeten Zeichenketten, z.B. "Hurra !", sind Stringkonstante. Eine Stringvariable existiert in C nicht! Strings werden in einem char-Feld abgelegt, z.B.:

```
char cfeld[8];   /* Feld mit 8 Elementen Typ char */
```

Als Endekennung wird das Bitmuster 0, also das Sonderzeichen mit der Ordnungszahl 0 verwendet ('\0').

Eine Zuweisung von ganzen Feldern ist aber nicht möglich, damit können auch Strings nicht zugewiesen werden. Es können nur die einzelnen Elemente, die Zeichen, transferiert werden. Für die Manipulation von Strings gibt es daher eine Reihe von Bibliotheksfunktionen. Mehr dazu erfahren Sie in Kapitel 10.6.

Als Sonderregelung kann ein char-Feld bei seiner Vereinbarung mit einer Stringkonstanten vorbelegt (initialisiert) werden. Unser Feld char cfeld[8]; kann damit mit einer Zeichenkette vorbelegt werden:

```
char cfeld[8] = "Hurra !";
```

Dies ist keine Zuweisung, sondern eine Vorbelegung! Es ist daher nicht dasselbe, wenn man

```
char cfeld[8];
cfeld = "Hurra !";   /* falsch!! */
```

schreibt. Bereits in Kapitel 3.4.4 wurde darauf hingewiesen, dass Zuweisung und Initialisierung zwei verschiedene Dinge sind, bei Strings muss man dies stets beachten.

In dem vorbelegten `char`-Feld stehen nun die 7 Zeichen `"Hurra !"` und im letzten Element die Endekennung, der Wert 0. Unser Feld war also gerade groß genug für diesen Text! In anderen Programmiersprachen werden Zeichenketten oft mit einer Längenangabe plus der eigentlichen Zeichenkette gespeichert. In C wird die Länge nicht mit abgespeichert, sondern nur das Ende des Strings (eben mit dem Null-Wert) gekennzeichnet. Man nennt dies Null-Terminiert.

Mit den bekannten Operationen können nun die einzelnen Zeichen in dem Feld (Array) verändert oder ausgegeben werden.

Wie können nun einzelne Zeichen der Zeichenkette ausgegeben werden?

```
printf("%c\n", cfeld[0]);
cfeld[5] = 'y';
printf("%s\n", cfeld);
```

Man kann ganze Zeichenketten natürlich auch von der Tastatur einlesen: `scanf` kennen Sie ja bereits aus dem Kapitel 4.

```
char cfeld[20];
scanf("%s", cfeld);
printf("Ihre Eingabe war: %s \n", cfeld);
```

Aber dieser Aufruf ist gefährlich: da in C ja keine Überprüfung der Grenzen von Arrays stattfindet, könnte ein Benutzer Ihres Programms einfach mehr als 19 (nicht 20 !) Zeichen eingeben. Die Größe des Arrays `cfeld` ist hierfür eigentlich nicht ausreichend, aber dies wird von `scanf` in der Form nicht erkannt. Die sichere Variante dieses Aufrufs sei auch genannt (ohne sie hier näher zu erklären):

```
scanf("%19s", cfeld);
```

Bemerkung: Da es sich bei einem String um eine Folge von beliebigen Zeichen handelt, ist hier eine Abfrage nach dem richtigen Datentyp nicht nötig.

8.2.3 Häufige Fehler

- Die Indices bei Arrays beginnen bei 0 und enden bei Größe-1. Folgendes wäre also falsch: `int x[5]; x[5]=25;`
- Eine Zeichenkette enthält immer ein Zeichen mehr als ausgegeben werden: die `'\0'` (binäre Null!) schließt die Zeichenkette ab! Der String `"Hi"` benötigt also drei Speicherzellen.

- Bei Ein- und Ausgabe von Strings muss der Feldname, nicht ein Feldelement (Zeichen) angegeben werden.

8.3 Aufzählungstyp (`enum`)

Bisher haben wir die Standard-Datentypen in C kennengelernt, die man in zwei Gruppen einteilen kann:

Die ganzzahligen Datentypen

- `int` mit den Varianten `long`, sowie `unsigned` für den positiven Zahlenbereich,
- `char` für Zeichen, die entsprechend der Kodierung eine Teilmenge der ganzzahligen Datentypen darstellen.

Die Gleitkomma-Datentypen

- `float` und `double`

Von allen diesen Datentypen können als abgeleiteter Datentyp Felder erzeugt werden.

Man kann aber auch selber einen neuen, unstrukturierten Datentyp vereinbaren. Dies geschieht durch Aufzählung der Menge der Werte, die zu diesem Typ gehören sollen (daher auch Aufzählungstyp oder Skalartyp genannt).

Die folgenden Typvereinbarungen sind Beispiele hierfür:

```
enum Tag {Mo,Di,Mi,Do,Fr,Sa,So};
enum Geschlecht {m,w};
enum Farbe {blau,rot,gruen,weiss,gelb,schwarz};
```

Der Wert `rot` ist also vom Typ `enum Farbe`, und `Mo` ist vom Typ `enum Tag`. Mit der Aufzählung aller möglichen Werte, die dieser Typ annehmen kann, wird ein neuer Aufzählungstyp vereinbart. Damit ist nur ein neuer Datentyp, aber noch keine Variable vereinbart. Die Vereinbarung des Aufzählungstyps muss mit einem Strichpunkt abgeschlossen werden.

Mit der Variablenvereinbarung

```
enum Tag t;
```

wird eine Variable `t` vom Typ `enum Tag` deklariert, der Sie z.B. den Wert `Mo` zuweisen können.

Bei Aufzählungstypen sind natürlich nur Wertzuweisungen möglich, die bei der Typvereinbarung angegeben sind:

```
enum Tag {Mo,Di,Mi,Do,Fr,Sa,So};
enum Tag s, t;
t = Mo;
s = Di;
t = s;
```

`enum Tag` ist der neue Datentyp, der nur die in geschweiften Klammern `{ }` angegebenen Werte annehmen kann. Von diesem Datentyp `enum Tag` können nun beliebige Variablen angelegt werden, denen die angegebenen Werte zugewiesen werden können. Anhand des Typs der neuen Variable ist auch klar, welche Werte sie annehmen kann: hier eben `Mo`, `Di`,... `So`.

Neben Wertzuweisungen sind bei Aufzählungstypen auch Vergleiche möglich: Die Werte eines Aufzählungstyps sind geordnet, wobei der erste aufgeführte Wert der „kleinste“ und der letzte der „größte“ ist. Bei Aufzählungstypen wird generell dem ersten Wert der Wert 0, dem nächsten der Wert 1 usw. zugeordnet. Damit entspricht der Aufzählungstyp einer Teilmenge der positiven Zahlen.

Beim Typ

```
enum T {x1,x2, ... xN};
```

hat `x1` den Wert 0, `xN` den Wert `N-1`.

Beispiel: Wir definieren den Typ `enum Mon` als

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

Welcher Wert ergibt sich für `Dez`? 11, da bei 0 zu zählen begonnen wird.

Es handelt sich immer um einen Unterbereich der positiven Zahlen von 0...n. Variablen eines Aufzählungstyps können in allen Operationen verwendet werden, in denen ganzzahlige Werte vorkommen können, beispielsweise als Schleifenvariable oder `case`-Label in einer `switch`-Anweisung. Zu beachten ist aber, dass eine Überwachung des eingeschränkten Wertebereiches nicht erfolgt, dies bleibt dem Programmierer vorbehalten.

Da es in C keinen booleschen Datentypen gibt, könnte man sich einen Aufzählungstyp

```
enum BOOLEAN {FALSE,TRUE};
```

definieren, damit ist auch die Gültigkeit von FALSE < TRUE vorgegeben. Da FALSE der Wert 0 und TRUE der Wert 1 zugeordnet wird, können Variable des Typs `enum BOOLEAN` wie logische Variable verwendet werden, z.B. in einer `if`-Verzweigung.

```
#include <stdio.h>          /* nötig für Ein-/Ausgabe */

int main()                  /* Programm-Kopf */
```

```
{
  enum Tag {Mo,Di,Mi,Do,Fr,Sa,So};
  enum Tag s, t;

  t = Mo;
  s = Di;
  if (Mo < Sa)
    printf ("Montag kommt vor Samstag!\n");
  printf ("Achtung: enum Ein-/Ausgabe nur mit Zahlen: s entspricht %d\n", s);
  return 0;
}/* main */
```

Listing 8.4: enum
Das Programm zeigt, wie die Verwendung von `enum` das Programmieren übersichtlicher gestaltet. Das Programm zeigt aber auch, dass `enum` nur als Zahl eingelesen, bzw. ausgegeben werden kann (siehe 2. `printf`-Anweisung).

8.4 Datenstyp Struktur (`struct`)

In einem Array sind alle Elemente vom selben Typ. Nun kommen wir zum Typ Struktur, dessen Daten aus einer festen Anzahl von Komponenten bestehen, die aber verschiedenartige Typen haben können.

Beispiel: Das Datum besteht aus Tag, Monat und Jahr. Wir können dafür eine Struktur (auch Verbund genannt) definieren:

```
enum Monat {                  /* -                    */
  Jan,Feb,Mrz,Apr,Mai,Jun,    /* | Aufzählungstyp     */
  Jul,Aug,Sep,Okt,Nov,Dez     /* |                    */
};                            /* -                    */

struct Datum {                /* -                          */
  int tag;                    /* |                          */
  enum Monat mon;             /* | Struktur-Vereinbarung    */
  unsigned int jahr;          /* |                          */
};                            /* -                          */

struct Datum d1,d2;           /* Variablendeklaration, Typ: Datum */
```

`struct Datum` ist der neue Datentyp, der aus den Komponenten

```
int tag;
enum Monat mon;
unsigned int jahr;
```

besteht. Die beiden Variablen `d1` und `d2` bestehen damit auch jeweils aus diesen drei Komponenten.

Genau wie beim Aufzählungstyp muss auch beim Strukturtyp die Vereinbarung mit einem Strichpunkt abgeschlossen werden.

Auf die einzelnen Komponenten der Struktur kann man durch die sogenannte Strukturkomponente zugreifen:

```
Strukturkomponente ::= Strukturvariable " . " Komponentenname .
```

Beispiele:

```
d1.tag d2.tag
d1.mon d2.mon
d1.jahr d2.jahr
```

Mit den Strukturkomponenten sind natürlich auch Wertzuweisungen möglich, z.B.:

```
d1.tag=18; d2.tag=31;
d1.mon=Jun; d2.mon=Jul;
d1.jahr=1988; d2.jahr=2002;
```

Damit repräsentieren die Variablen `d1` und `d2` die beiden Daten 18. Juni 1988 und 31. Juli 2002.

Die Syntax der Struktur-Vereinbarung geht aus unserem Beispiel hervor. Strukturen oder Verbunde haben die Eigenschaft, dass die Komponenten selbst strukturiert sein dürfen. Strukturkomponenten können also auch Felder sein oder selbst wieder vom Typ Struktur. Sehen wir uns dazu den nächsten Abschnitt an.

Beispiel: Eine „Person“ lässt sich durch Name, Geschlecht und Geburtsdatum beschreiben:

```
enum Art {m,w};
struct Person
{
  char vorname[31], nachname[31];
  enum Art geschlecht;
  struct Datum geburtsdatum;
};
struct Person pers1;
```

Den Datentyp `struct Datum` übernehmen wir aus dem vorangegangenen Beispiel. Der Variablen `pers1` soll eine bestimmte Person zugeordnet sein. Übrignes kann man das Vereinbaren des Strukturtyps und das Definieren einer Variable von diesem neuen Strukturtyp auch zusammenfassen:

```
struct Person
{
  char vorname[31], nachname[31];
  enum Art geschlecht;
  struct Datum geburtsdatum;
} pers1;
```

Wir wollen unser Beispiel nun auf eine Schulklasse anwenden. Dazu verwenden wir den Typ `Person` für jeden Schüler und erweitern den Vereinbarungsteil um das Feld `klasse` für 40 Schüler.

```
enum Monat {Jan,Feb,Mrz,Apr,Mai,Jun,Jul,Aug,Sep,Okt,Nov,Dez};
struct Datum
{
  int tag;
  enum Monat mon;
  unsigned int jahr;
};

enum Art {m,w};
struct Person
{
  char vorname[31], nachname[31];
  enum Art geschlecht;
  struct Datum geburtsdatum;
};
struct Person schueler, klasse[40];
```

Das Feld klasse hat 40 Werte des Strukturtyps `struct Person`.

Durch welche Komponentenvariable können Sie auf den Anfangsbuchstaben des Nachnamens des ersten Schülers des Feldes zugreifen? Der erste Schüler im Feld `klasse` ist `klasse[0]`, sein Nachname dann `klasse[0].nachname` und davon der erste Buchstabe `klasse[0].nachname[0]`.

Wie sieht die entsprechende Variable für das Geburtsjahr dieses Schülers aus? Hier kann man die „Verschachtelung“ der Strukturen durch Aneinanderreihen der Komponentenzugriffe auflösen; man erreicht das Geburtsjahr über die folgende Komponentenvariable:

```
klasse[0].geburtsdatum.jahr
```

```
#include <stdio.h>                    /* nötig für Ein-/Ausgabe */

int main()                            /* Programm-Kopf */
{
  int i, j;
  enum Monat{Jan,Feb,Mrz,Apr,Mai,Jun,Jul,Aug,Sep,Okt,Nov,Dez};/*Aufzählungstyp*/
  enum Art {w,m};
  enum Tag {Mo,Di,Mi,Do,Fr,Sa,So};
  struct Datum /* Struktur-Vereinbarung */
  {
    int tag;
    enum Monat mon;
    unsigned int jahr;
  };

  struct Person /* Struktur-Vereinbarung */
  {
    char vorname[31], nachname[31];
    enum Art geschlecht;
    struct Datum geburtsdatum;
  };

  struct Person persStruct;

  printf ("Vorname? ");
  if (scanf("%s", persStruct.vorname)==0)
  {
      printf ("Unzulaessige Eingabe!");
      return 1;
  }/*funktioniert nur bis zu einer Länge von 30->Arraygröße!*/
  printf ("Nachname? ");
  if (scanf("%s", persStruct.nachname)==0)
  {
      printf ("Unzulaessige Eingabe!");
      return 1;
  }/*funktioniert nur bis zu einer Länge von 30->Arraygröße!*/
  printf ("Geburtstag? ");
  if (scanf(" %d", &persStruct.geburtsdatum.tag)==0)
  {
      printf ("Unzulaessige Eingabe!");
      return 1;
  }
  printf ("Geburtsmonat?"); /*Benutzer sollte 1-12 eingeben*/
  if (scanf(" %d", &j)==0)
  {
      printf ("Unzulaessige Eingabe!");
      return 1;
```

```
    }
    persStruct.geburtsdatum.mon = (enum Monat)(j-1); /*enum Monat kennt Werte von 0-11*/
    printf ("Geburtsjahr? ");
    if (scanf(" %u", &persStruct.geburtsdatum.jahr)==0)
    {
        printf ("Unzulaessige Eingabe!");
        return 1;
    }
    printf ("Geschlecht [weiblich: 1, maenlich: 2]? "); /*Benutzer sollte 1(weiblich)
                    oder 2(männlich) eingeben*/
    if (scanf(" %d", &j)==0)
    {
        printf ("Unzulaessige Eingabe!");
        return 1;
    }
    persStruct.geschlecht = (enum Art)(j-1); /*enum Art kennt Art von 0-1*/

    printf ("Folgende Daten wurden eingegeben:\n");
    if (persStruct.geschlecht == w)
      printf ("Frau\n");
    else
      printf ("Herr\n");
    if (persStruct.geburtsdatum.mon < Jul)
      printf ("%s %s wurde in der ersten Jahreshaelfte (%d.%d.) %d geboren.\n",
        persStruct.vorname, persStruct.nachname, persStruct.geburtsdatum.tag,
        persStruct.geburtsdatum.mon+1, persStruct.geburtsdatum.jahr);
    else
      printf ("%s %s wurde in der zweiten Jahreshaelfte (%d.%d.) %d geboren.\n",
        persStruct.vorname, persStruct.nachname, persStruct.geburtsdatum.tag,
        persStruct.geburtsdatum.mon+1, persStruct.geburtsdatum.jahr);
    return 0;
}/* main */
```

Listing 8.5: struct
Das Programm zeigt, wie Strukturen in C verwendet werden. Beachten Sie bitte, dass die Strings `vorname` und `nachname` nur 31 Zeichen aufnehmen können, wobei das 31. jeweils zum Abschluss des Strings dient. Gibt der Benutzer mehr als 30 Zeichen ein, so kann es zu ungewöhnlichem Verhalten des Programms kommen, weil unbestimmte Speicherbereiche beschrieben werden. Siehe hierzu Kap. 8.2.2 und 8.2.3.

8.5 Felder von zusammengesetzten Datentypen

Wir haben in diesem Kapitel zusammengesetzte Datentypen kennen gelernt. In Form von mehrdimensionalen Feldern haben wir auch schon mit Feldern von zusammengesetzten Datentypen gearbeitet. Ein zweidimensionales Feld kann als Array von eindi-

mensionalen Arrays gesehen werden. Konkret könnte das aus Kapitel 8.1.4 stammende zweidimensionale Integerfeld

```
int matrix[100][50];
```

als ein Feld von 100 Feldern, die je 50 Integerzahlen enthalten, verstanden werden. Dementsprechend können auch von anderen zusammengesetzten Datentypen auf naheliegende Weise Felder angelegt werden. Dies wird anhand der bereits in *struct* verwendeten Struktur `Person` in *struct mit Schleifen* demonstriert. Es wird ein Feld namens `leute` angelegt, das zweimal die Struktur `person` enthält. Er kann nun über den Index von `leute` auf die jeweils gewünschte Struktur zugegriffen werden. Dies kann insbesondere dann sehr hilfreich sein, wenn man bestimmte Operationen für mehrere Strukturvariablen ausführen will. So kann im Beispiel das Einlesen und Ausgeben für beide Strukturvariablen in `leute` effizient in `for`-Schleifen realisiert werden.

```
#include <stdio.h>                    /* nötig für Ein-/Ausgabe */

int main()                            /* Programm-Kopf */
{
  int i, j;
  enum Monat{Jan,Feb,Mrz,Apr,Mai,Jun,Jul,Aug,Sep,Okt,Nov,Dez};/*Aufzählungstyp*/
  enum Art {w,m};
  enum Tag {Mo,Di,Mi,Do,Fr,Sa,So};
  struct Datum /* Struktur-Vereinbarung */
  {
   int tag;
   enum Monat mon;
   unsigned int jahr;
  };

  struct Person /* Struktur-Vereinbarung */
  {
    char vorname[31], nachname[31];
    enum Art geschlecht;
    struct Datum geburtsdatum;
  };

  struct Person leute[2];  /* Feld mit zwei Elementen, die vom Typ Struktur sind */

  for(i=0; i < 2; i++)
    {
      printf ("Vorname %d? ", i+1);
      if (scanf("%s", leute[i].vorname)==0)
      {
          printf ("Unzulaessige Eingabe!");
          return 1;
      }/*^-- funktioniert nur bis zu einer Länge von 30->Arraygröße!*/
```

```
        printf ("Nachname %d? ", i+1);
        if (scanf("%s", leute[i].nachname)==0)
        {
            printf ("Unzulaessige Eingabe!");
            return 1;
        }/* ^-- funktioniert nur bis zu einer Länge von 30->Arraygröße!*/
        printf ("Geburtstag %d? ", i+1);
        if (scanf(" %d", &leute[i].geburtsdatum.tag)==0)
        {
            printf ("Unzulaessige Eingabe!");
            return 1;
        }
        printf ("Geburtsmonat %d? ", i+1); /*Benutzer sollte 1-12 eingeben*/
        if (scanf(" %d", &j)==0)
        {
            printf ("Unzulaessige Eingabe!");
            return 1;
        }
        leute[i].geburtsdatum.mon = (enum Monat)(j-1); /*enum Monat kennt Werte von 0-11*/
        printf ("Geburtsjahr %d? ", i+1);
        if (scanf(" %u", &leute[i].geburtsdatum.jahr)==0)
        {
            printf ("Unzulaessige Eingabe!");
            return 1;
        }
        printf ("Geschlecht %d [weiblich: 1, maenlich: 2]? ", i+1); /*Benutzer sollte
1(weiblich)
                                                    oder 2(männlich)
eingeben*/
        if (scanf(" %d", &j)==0)
        {
            printf ("Unzulaessige Eingabe!");
            return 1;
        }
        leute[i].geschlecht = (enum Art)(j-1); /*enum Art kennt Art von 0-1*/
    }/*for*/

  printf ("Folgende Daten wurden eingegeben:\n");
  for(i=0; i <2; i++)
    {
      if (leute[i].geschlecht == w)
  printf ("Frau\n");
      else
  printf ("Herr\n");
      if (leute[i].geburtsdatum.mon < Jul)
  printf ("%s %s geboren in der ersten Jahreshaelfte (%d.%d.) %d.\n",
          leute[i].vorname, leute[i].nachname, leute[i].geburtsdatum.tag,
```

```
            leute[i].geburtsdatum.mon+1, leute[i].geburtsdatum.jahr);
        else
      printf ("%s %s geboren in der zweiten Jahreshaelfte (%d.%d.) %d.\n",
            leute[i].vorname, leute[i].nachname, leute[i].geburtsdatum.tag,
            leute[i].geburtsdatum.mon+1, leute[i].geburtsdatum.jahr);
    }/* for */
    return 0;
}/* main */
```

Listing 8.6: struct mit Schleifen
Das Programm verdeutlicht, wie man ein Feld von Strukturen im Zusammenhang mit Schleifen verwenden kann, um auf einzelne Strukturen, bzw. deren Elemente zugreifen zu können.

9 Zeiger

9.1 Zeiger (pointer)

Wir haben bislang immer von Variablen gesprochen, die in einer (oder mehreren) Zelle(n) im Speicher abgelegt sind. Es gibt in C aber noch einen weiteren Weg, auf den Inhalt von Speicherzellen zuzugreifen: über Zeiger (Pointer). Ein Zeiger ist nichts weiter als eine besondere Variable, in der die Adresse einer (anderen) Speicherzelle abgelegt ist.

Den Programmierer interessiert die Adresse einer Speicherzelle normalerweise nicht, da er mit den symbolischen Namen, den Variablen, arbeiten kann. Unter bestimmten Umständen ist es aber durchaus sinnvoll und eleganter mit den Adressen, also mit Zeigern zu arbeiten. Dies ist z.B. dann der Fall, wenn nicht von vornherein klar ist, wieviele Daten ein Programm im Speicher ablegen muss und daher „normale“ Variablen nicht ausreichend sind: in diesem Fall werden dynamische Datenstrukturen eingesetzt (siehe Kapitel 11.1), bei denen die Speicherzellen über ihre Adresse angesprochen werden müssen. Daneben werden Zeiger auch häufig bei Feldern und bei der Übergabe von Parametern an Funktionen eingesetzt.

In C existieren für Zeiger zwei Operatoren:

- `&` (Adressoperator)
- `*` (Zeiger- oder Dereferenzierungsoperator)

Mit dem `&`-Operator wird die Adresse einer Variablen ermittelt. Ist die Variable mit `int i;` vereinbart, so ist `&i` dann die Adresse der Zelle `i`, `&i` ist also der Zeiger auf `i`.

Jede Variable hat einen Datentyp. Bei der Vereinbarung der Variablen muss angegeben werden, welchen Typ die dort abgelegten Daten haben. In unserem Beispiel ist `i` vom Typ `int`. Damit ist `&i` ein „Zeiger auf `int`“. Im Beispiel `double d;` wäre `&d` der Zeiger auf `d`, also ein „Zeiger auf double“.

Diesen Wert vom Typ „Zeiger auf...“ möchte man nun gerne in einer Variablen ablegen (die Adresse allein bringt uns ja noch nicht weiter). Dies geht nicht in einem der bisher bekannten Standard-Datentypen. Es muss der Datentyp „Zeiger auf...“ verwendet werden. Der Typ des Zeigers entscheidet, wie der Inhalt der Speicherzelle, auf die der Zeiger zeigt, zu interpretieren ist. Der Zeiger selbst ist immer nur eine Adresse. Die Deklaration für eine Zeigervariable ist:

```
int *iptr;
double *dptr1, *dptr2;
```

`iptr` ist eine Variable vom Typ „Zeiger auf `int`“, `dptr1` und `dptr2` sind vom Typ „Zeiger auf `double`". Beachten Sie, dass bei der Variablendeklaration vor jeder Zeigervariable ein `*` stehen muss! Fehlt das `*` Zeichen, so wird eine normale Variable deklariert - man kann also Zeiger und normale Variablen in einer Zeile mischen. Schreibt man z.B.

```
double d, *dptr;
```

so ist `dptr` ein „Zeiger auf `double`“, während `d` eine gewöhnliche `double`-Variable ist. Wird nun dieser Zeigervariable die Adresse einer passenden Variable zugewiesen:

```
iptr = &i;
dptr1 = &d;
```

so ist `iptr` ein Zeiger auf `i` (Typ `int`) und `dptr1` ein Zeiger auf `d` (Typ `double`).

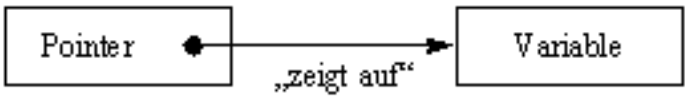


Abbildung 9.1: Zeiger auf eine Variable

Der zu diesem Adressoperator `&` inverse Operator ist der Zeigeroperator `*` (auch Dereferenzierungsoperator genannt). Mit dem Zeigeroperator kann man über den Zeiger auf den Inhalt einer Variable zugreifen.

```
*iptr = 123;
```

greift auf die Zelle zu, auf die iptr zeigt. Dies ist daher identisch mit

```
i = 123;
```

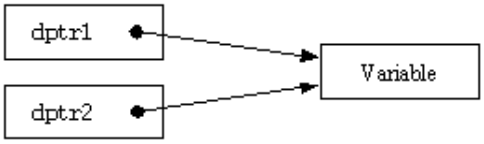
unter der Voraussetzung, dass mit

```
iptr = &i;
```

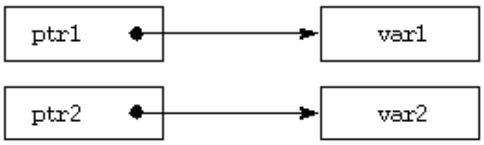
die Zeigervariable richtig belegt wurde.
Zeigervariable des gleichen Typs können auch einander zugewiesen werden. Die Zuweisung

```
dptr2 = dptr1;
```

bewirkt:

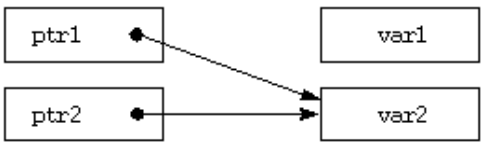


`dptr1` und `dptr2` zeigen auf dieselbe Bezugsvariable, auf den gleichen Speicherplatz. Dies hat natürlich zur Folge, dass eine Änderung von „Variable“ über beide Pointer möglich ist, und jede Änderung entsprechend auch über *beide* Zeiger sichtbar wird. Dies ist nicht zu verwechseln mit folgender Situation:

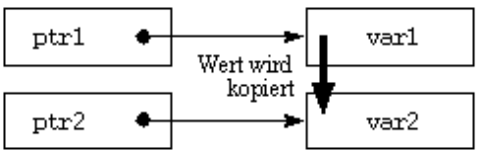


Hier zeigen die beiden Pointer auf verschiedene Variablen. Auch wenn der Inhalt der Variablen identisch ist, so sind die Zeiger selbst *nicht* gleich, sie zeigen ja auf unterschiedliche Speicherzellen. Zwei Pointervariablen sind nur gleich, wenn sie auf die gleiche Bezugsvariable zeigen.

Wie kann man in folgendem Beispiel der `var2` den Wert von `var1` zuweisen, wenn nur die beiden Zeiger `ptr1` und `ptr2` bekannt sind? Der einfache Versuch `ptr1 = ptr2` bewirkt:



Jetzt zeigen `ptr1` und `ptr2` auf die selbe Bezugsvariable. Der Inhalt der Variablen wurde nicht kopiert, `var2` hat immer noch ihren alten Wert. Besser ist folgender Versuch: `*ptr2 = *ptr1` :



Der Bezugsvariablen von `ptr2` wird der Wert der Bezugsvariablen von `ptr1` zugewiesen. Vorher muss natürlich beiden Zeigervariablen `ptr1` und `ptr2` je die Adresse ihrer Bezugsvariable zugewiesen werden.

```
#include <stdio.h>

int main()
{
  int var1, var2;
  int *ptr1,*ptr2;
  var1 = 3;
  var2 = 6;
  ptr1 = &var1;   /* ptr1 zeigt auf var1*/
  ptr2 = &var2;   /* ptr2 -- " -- var2*/
  printf ("1. var1 = %2d, *ptr1 = %2d, var2 = %2d *ptr2 = %2d\n", var1, *ptr1, var2,
*ptr2);
  var2 = var1;    /* var2 wird auf den Wert von var1 gesetzt*/
  printf ("2. var1 = %2d, *ptr1 = %2d, var1 = %2d *ptr2 = %2d\n", var1, *ptr1, var2,
*ptr2);
  var1 = 12;      /* var1 wird auf 12 gesetzt */
  printf ("3. var1 = %2d, *ptr1 = %2d, var1 = %2d *ptr2 = %2d\n", var1, *ptr1, var2,
*ptr2);
  *ptr2 = *ptr1;  /* der Wert der Variable auf die ptr2 zeigt,
                     // wird auf den Wert der Variable gesetzt,
                     // auf die ptr1 zeigt*/
  printf ("4. var1 = %2d, *ptr1 = %2d, var2 = %2d *ptr2 = %2d\n", var1, *ptr1, var2,
*ptr2);
  var1 = 24;      /* var1 wird auf 24 gesetzt */
  printf ("5. var1 = %2d, *ptr1 = %2d, var1 = %2d *ptr2 = %2d\n", var1, *ptr1, var2,
*ptr2);
  ptr2 = ptr1;    /* ptr2 zeigt nun auf den gleichen Speicherbereich wie ptr1 */
  printf ("6. var1 = %2d, *ptr1 = %2d, var1 = %2d *ptr2 = %2d\n", var1, *ptr1, var2,
*ptr2);
  return 0;
}/* main */
```

Listing 9.1: Zuweisung von Zeigern

Nach den obigen Erklärungen demonstriert das Programm die verschiedenen Zuweisungen von Variablen, bzw. Zeigern. Übersetzen Sie das Programm und führen es anschließend aus. Machen Sie sich anhand der Ausgaben des Programms, der Kommentare im Quelltext und der oben erbrachten Erklärungen klar, was in diesem Programm passiert.

9.2 Zeiger und Arrays

Das Arbeiten mit Zeigern wird dann besonders interessant, wenn Felder bearbeitet werden. Mit der Vereinbarung

```
int feld[20], *pi;
```

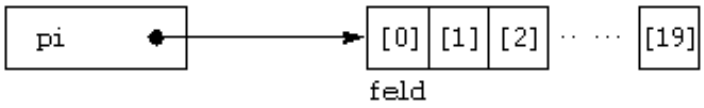
wird ein Feld mit 20 Elementen angelegt und gleichzeitig eine Variable vom Typ „Zeiger auf `int`“. Der Inhalt der Zelle `pi` ist ebenso wie der Inhalt der Feldelemente undefiniert.

```
pi = &feld[0];
```

weist der Pointervariablen die Adresse des 1. Feldelementes (Index 0) zu. Diese Zuweisung kann man in C auch kürzer schreiben: Da der Feldname ohne einen Index ohnehin die Adresse des Feldes, also des 1. Elementes ist, kann dies auch mit

```
pi = feld;
```

erreicht werden.



Mit `*pi` kann man nun auf das erste Feldelement zugreifen, folgende zwei Zeilen haben also die selbe Wirkung:

```
*pi = 123;
feld[0] = 123;
```

Zeigervariablen kann man zuweisen, vorausgesetzt, sie sind vom gleichen Typ. Man kann sie auch auf Gleichheit überprüfen (`==` , `!=`) und miteinander vergleichen (>,>=,<,<=). Solche Vergleiche machen aber nur einen Sinn, wenn beide Zeiger auf Elemente des gleichen Feldes zeigen!

Mit Zeigervariablen kann man auch rechnen; als Operationen sind erlaubt:

```
pi++
pi--
pi = pi + n
```

`pi++` inkrementiert die Zeigervariable `pi`, setzt den Zeiger also auf das nächste Feldelement. Die Addition eines Offsets bewegt den Zeiger um `n` Elemente weiter. Solange die Feldgrenzen nicht über- oder unterschritten werden, kann man damit auf beliebige Elemente des Feldes zugreifen.

Diese Zeigerarithmetik gilt für alle Felder mit beliebigen Datentypen. Der Zeiger wird unabhängig von der Größe der Elemente (`char`,`int`,`double`,`struct`,...) jeweils zum nächsten, bzw. `n`-ten Element weitergeschaltet. Je nach Zeigertyp ändert sich die Adresse also unterschiedlich stark! Es gibt also zwei wichtige Gründe, warum Zeiger auch einen Typ haben, obwohl sie ja „nur“ auf eine Speicherzelle zeigen:

- Für die Dereferenzierung (Zugriff auf den Wert der Speicherzelle, auf die der Zeiger zeigt: `i = *iptr`) muss bekannt sein, wie der Inhalt der Speicherzelle zu interpretieren ist.
- Bei der Zeigerarithmetik bedeutet ein Inkrement um 1 das Weiterstellen des Zeigers zur nächsten Variable. Dazu muss die Größe der Variable zur Adresse addiert werden. Die Größe wiederum hängt vom Typ ab.

9.3 Zeiger und lokale Variablen

Man kann natürlich auch die Adresse von lokalen Variablen ermitteln und damit einen Zeiger auf sie zeigen lassen. Bei statischen lokalen Variablen geht das noch ganz gut, deren Adresse ändert sich ja nicht. Bei automatischen lokalen Variablen kann es aber sehr schnell schief gehen, da nach dem Ende der Funktion diese Speicherzelle wieder freigegeben wird und für andere Aufgaben verwendet werden kann. Eine (automatische) lokale Variable existiert nur innerhalb ihres Blocks!

Nach dem Ende der Funktion gibt es die Variable dann einfach nicht mehr. Ein Zeiger auf sie würde nun auf freien oder irgendwann auch wieder anderweitig benutzten Speicher zeigen.

```
main()
{
  int feld[20], *pi, i;

  for(i = 0; i < 20; i++)
  {
    feld[i] = i + 1;
  }

  pi = feld;    /* Pi zeigt auf das erste Element von feld[]; */

  while(*pi != 20)
  {
    if (*pi % 2) /*immer wenn die integer-Zahl, auf die pi zeigt, ungerade ist... */
      *pi = -(*pi);  /*wird diese integer Zahl negiert */
    pi++;  /* pi soll nun auf den nächsten integer-Wert zeigen*/
  }

  printf("Wir ueberpruefen nun, ob der Zugriff auf das Feld mit Zeigern erfolgreich war...\n");
  for(i = 0; i < 20; i++)
  {
    printf("Wert von feld[%d]\t= %3i\n", i, feld[i]);
  }
 return 0;
}/*main*/
```

Listing 9.2: Zeigerzugriff

Das Programm demonstriert den Zugriff auf das Feld `feld[20]` mittels des Zeigers `pi`. In diesem Beispiel wird über den bekannten Inhalt des Feldes sichergestellt, dass dessen Grenzen beim Zugriff durch einen Zeiger nicht überschritten werden. Dies geschieht dadurch, dass die `while`-Schleife abbricht, sobald der Zeiger `pi` auf die in Durchlaufrichtung letzte Integerzahl im Feld feld zeigt, deren Wert gleich 20 ist.

Fv

10 Funktionen

10.1 Unterprogramme, Lokalität

10.1.1 Funktionen

Mit Unterprogrammen (auch Prozeduren oder Funktionen genannt) kann man Teile eines Programms unter einem eigenen Namen zusammenfassen. Ein Unterprogramm kann man sich so vorstellen :

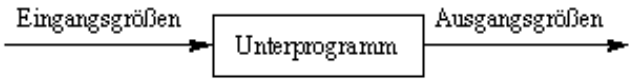


Abbildung 10.1: Funktionsweise eines Unterprogramms

Dem Unterprogramm werden Eingangsgrößen übergeben, aus denen es dann die Ausgangsgrößen ermittelt. Das Unterprogramm ist also ein Teilprogramm, in dem Teilprobleme abgehandelt werden. Immer wenn in einem Programmanlauf ein solches Teilproblem auftaucht, kann der Programmierer das entsprechende Unterprogramm benutzen. Unterprogramme spielen eine grundlegende Rolle beim Aufbau des Programms, denn damit kann es in übersichtlicher Weise strukturiert werden, was vor allem bei langen und komplexen Algorithmen vorteilhaft ist. Wenn das Unterprogramm öfter aufgerufen wird, verringert sich damit natürlich auch die Länge des Programms, da das Unterprogramm nur einmal vorhanden ist, aber öfters benutzt wird.

Im Gegensatz zu anderen Programmiersprachen gibt es in C nur einen Unterprogrammtyp, die Funktion. Wir werden daher im folgenden nur noch von Funktionen sprechen.

Funktionen sind aus der Mathematik bekannt. Die Funktion `sin(wi)` hat für einen gegebenen Winkel `wi` einen bestimmten Wert, sie liefert einen Wert zurück. Damit erklären wir den Begriff Funktion folgendermaßen :

Eine Funktion ist ein Unterprogramm, das aus einer oder mehreren Eingangsgrößen einen Wert berechnet und diesen zurückliefert.

In die Programmierung übertragen, bedeutet dies, dass die Funktion abhängig von Eingangsgrößen einen Wert berechnet. Jeder Wert hat einen bestimmten Typ, damit hat auch die Funktion einen bestimmten Typ, nämlich den Typ des zurückgegebenen

Wertes. Die Funktion `sin(wi)` berechnet und liefert einen Gleitkommawert, d.h., die Funktion `sin` ist vom Typ `double`.

Achtung: Man kann in C durchaus auch Funktionen schreiben, die keine Eingangsgrößen und/oder keine Ausgangsgrößen haben. Ein sinnvolles Beispiel wäre eine Funktion, die einfach nur bei jedem Aufruf einen bestimmten Text auf dem Bildschirm ausgibt. Diese Funktion kann man dann auch an beliebigen Programmstellen benutzen, sie braucht aber keine Eingangsgrößen und hat auch keine Ausgangsgrößen (die Programmausgabe zählt nicht als Ausgangsgröße).

Hier ein einfaches Beispiel für eine Funktionsvereinbarung zur Berechnung des Mittelwerts eines Feldes `arr` vom Typ `float`. Der Mittelwert ist dann ebenfalls vom Typ `float`.

```
void mittelwert( )     /* Funktionskopf     */
{
  sum=0.0;             /* -                 */
  for (i=0; i<n; i++) /* | Funktionsblock */
    sum=sum+arr[i];    /* |                 */
  mw=sum/n;            /* -                 */
}
```

Welche Variable liefert dabei das gewünschte Ergebnis? Das Ergebnis steht am Ende in der globalen Variablen `mw`. Mehr dazu später, schauen wir uns erst mal an, wie dieses Beispiel aufgebaut ist: Wie man sieht, hat die Funktion den gleichen Aufbau wie ein Hauptprogramm (`main`). Die Funktion besteht zunächst aus einem Funktionskopf :

```
void mittelwert()
```

Hier wird der Funktionstyp angegeben, der Typ des Rückgabewertes. Da unser Beispiel keinen Rückgabewert haben soll, muss als Typ `void` angegeben werden. Wir werden im Kapitel 10.2.1 sehen, wie Funktionen mit Rückgabewert aussehen.

Auf den Funktionskopf folgt der Funktionsblock :

```
{
  sum=0.0;
  for (i=0; i<n; i++)
    sum=sum+arr[i];
  mw=sum/n;
}
```

Der Funktionsblock besteht wie ein Hauptprogramm aus einer Reihe von Anweisungen, die in geschweiften Klammern eingeschlossen sind. Es fehlen aber noch die Vereinbarungen der verwendeten Variablen.

10.1.2 Globale und lokale Variablen

Bisher haben wir Variablen nur innerhalb eines Blockes (dem Hauptprogramm `main`) vereinbart. Variablen, die innerhalb eines Blockes (ein Block wird immer von geschweiften Klammern eingeschlossen!) vereinbart wurden, sind auch nur innerhalb dieses Blockes gültig. Man nennt sie *lokale Variablen*. Variablen, die außerhalb aller Blöcke, also *extern* vereinbart werden, sind in *allen* Blöcken sichtbar und gültig, damit haben wir eine *globale Variable*.

Globale Variablen, die nicht vom Programmierer initialisiert werden, erhalten automatisch den Startwert 0.

Zwei Beispiele :

Beispiel (a)

```
float arr[100], mw, sum;
int i, n;

void mittelwert( )
{
  sum=0.0;
  for (i=0; i<n; i++)
    sum=sum+arr[i];
  mw=sum/n;
}
```

Beispiel (b)

```
float mw, arr[100];
int n;

void mittelwert( )
{
  float sum;
  int i;
  sum=0.0;
  for (i=0; i<n; i++)
    sum=sum+arr[i];
  mw=sum/n;
}
```

In unserem Beispiel (a) sehen wir, dass die Variablen `i` und `sum` nur innerhalb der Funktion `mittelwert` benötigt werden, während die Variablen `n`, `mw` und `arr` auch außerhalb von Bedeutung sind. Diese Lokalität der Variablen `i` und `sum` kann man durch eine Vereinbarung innerhalb des Funktionsblocks ausdrücken, wie dies im Fall (b) geschehen ist.

Ergänzen wir das Beispiel mit einem Hauptprogramm :

```
float mw, arr[100];
int n;

void mittelwert()
{
  float sum;
  int i;
  sum=0.0;
  for (i=0; i<n; i++)
    sum=sum+arr[i];
  mw=sum/n;
```

```
}

main()
{
  float x,y,z;
  int i,j,k;
  ...
}
```

Hier sind also `i` und `sum` lokale Variablen unserer Funktion `mittelwert`. Die Variablen `n` und das Feld `arr` sind außerhalb aller Blöcke, also extern, vereinbart und stehen damit in beiden Blöcken, `mittelwert` und `main`, zur Verfügung.

Lokale Variable haben nur einen begrenzten Existenzbereich. Sie sind außerhalb des Blocks, in dem sie vereinbart wurden, weder existent noch sichtbar, ihr Name ist dort also noch frei.

Das hat z.B. die Konsequenz, dass der gleiche Name für verschiedene Dinge verwendet werden kann. Das ist für den Programmierer sehr nützlich, da er lokale Namen beliebig verwenden kann, ohne Kenntnis der Namen, die schon irgendwo anders im Programm vorkommen. Die lokale Variable ist damit auch geschützt vor unerwünschten Veränderungen durch andere Programmteile.

Es ist guter Programmierstil, so wenige globale Variablen wie möglich zu vereinbaren. So werden Namenskonflikte vermieden und die Programme sind sehr viel übersichtlicher und besser lesbar.

Beachten Sie folgende Regeln zur Gültigkeit von Namen :

- Der Existenzbereich eines Namens ist der Block, in welchem er vereinbart ist und alle Blöcke innerhalb dieses Blocks.
- Wenn ein in einem Block vereinbarter Name `n` in einem Unterblock nochmals vereinbart ist, so gilt dieser neue Name in diesem Unterblock (und wiederum dessen Unterblöcken). Der Unterblock ist vom Gültigkeitsbereich des Namens `n` im Oberblock ausgeschlossen: der lokale Name im Unterblock überdeckt den Namen im Oberblock.

Also :

- Variablen, die innerhalb einer Funktion definiert sind, sind nur innerhalb dieser Funktion existent.
- Variablen, die außerhalb einer Funktion definiert sind, sind außen und innen existent.
- Variablen, die innen und außen mit gleichem Namen deklariert wurden, sind beide innen existent, aber nur das lokal vereinbarte Objekt ist innen sichtbar.

- Sichtbarkeit schließt Existenz ein.
- Sichtbar heißt, die Variable kann in der betreffenden Programmzeile angesprochen werden.

Die Funktionsdefinition hat die Form eines separaten Programms. Neben Variablenvereinbarungen können innerhalb der Funktionen alle anderen Vereinbarungen wie z.B. Typvereinbarungen vorkommen.

Zu beachten ist dabei, dass eine Funktion immer außerhalb des Programmes steht. Sie bildet einen eigenen Funktionsblock außerhalb aller anderen Programmblöcke. Eine Schachtelung von Funktionen, also Vereinbarung einer Funktion lokal innerhalb eines Blockes, wie dies in anderen Programmiersprachen möglich ist, gibt es in C nicht. Funktionen sind in C immer extern.

```
#include <stdio.h>     /* nötig für Ein-/Ausgabe */

float mw, arr[100];
int n;

void mittelwert( )     /* Funktionskopf    */
{
  int i ;
  float sum;

  sum=0.0;             /* -                */
  for (i =0; i <n; i ++) /* | Funktionsblock */
    sum=sum+arr[i];    /* |                */
  mw=sum/n;            /* -                */
}


int main()             /* Programm-Kopf */
{
  int i ;
  do
  {
    printf ("Bitte  Anzahl der Werte (<=100) eingeben! n = ");
    if (scanf("%d", &n)==0)
    {
        printf ("Unzulaessige Eingabe!");
        return 1;
    }
  }
  while(n > 100);

  printf ("Bitte  geben Sie nun die einzelnen Werte ein!\n");
  for(i  = 0; i  < n; i ++)
```

```
    {
      printf ("Wert %i: ", i+1);
      if (scanf("%f", &arr[i])==0)
      {
          printf ("Unzulaessige Eingabe!");
          return 1;
      }
    }

    mittelwert();
    printf ("\nEs ergibt sich ein Mittelwert von %f!\n\n", mw);
    return 0;

}/* main */
```

Listing 10.1: Mittelwertfunktion
Im Programm wird die Funktion `mittelwert()` verwendet. `main()` übernimmt die Ein- und die Ausgabe und ruft die Funktion `mittelwert()` auf, welche die eingentliche Berechnung übernimmt.

10.1.3 Statische Variablen

Das Konzept der lokalen Namen und die Regel, dass eine Variable nicht außerhalb ihres bestimmten Wirkungskreises existiert, haben zur Folge, dass ihr Wert verloren geht, wenn die entsprechende Funktion beendet ist. Wenn die Funktion später nochmals aufgerufen wird, ist ihr Wert wieder undefiniert. Wenn also die Variable ihren Wert zwischen zwei Funktionsaufrufen behalten soll, muss sie außerhalb der Funktion vereinbart werden.

Soll der Wert der Variablen nur lokal sichtbar sein, aber trotzdem von Aufruf zu Aufruf erhalten bleiben, so muss das Schlüsselwort `static` in der Deklaration der Variablen hinzugefügt werden.

Beispiel :

```
void zaehler()
{
  static int z=1;
  printf("Dies war der %d. Aufruf von zaehler\n", z);
  z++;
}
```

Ohne das Schlüsselwort `static` würde immer der Wert 1 ausgegeben, hier jedoch wird die Variable `z` nur einmal erzeugt und mit dem Wert 1 vorbelegt. Alle Änderungen bleiben erhalten, auch über mehrere Aufrufe der Funktion zaehler hinweg.

Das Gegenstück zu den `static`-Variablen nennt sich automatische Variablen (`auto`). Da diese aber den Normalfall darstellen, ist es nicht nötig, extra das Schlüsselwort `auto`

mit anzugeben. Für den Sprachgebrauch ist es aber hilfreich, von automatischen Variablen reden zu können, wenn man „nicht-statische Variablen“ meint.

Genau wie globale Variablen werden statische lokale Variablen mit dem Wert 0 vorbelegt, wenn sie nicht vom Programmierer explizit initialisiert werden.

Achtung: Der Default-Startwert von 0 gilt nur für globale und statische lokale Variablen! Alle automatischen lokalen Variablen (die aber den Normalfall darstellen) haben ohne Initialisierung einen undefinierten Wert, sie können also einen unbekannten, zufälligen Wert haben.

```
#include <stdio.h>          /* nötig für Ein-/Ausgabe */

void zaehler( )
{
  static int z=1;
  printf ("Dies war der %d. Aufruf von zaehler\n", z);
  z++;
}

int main()                  /* Programm-Kopf */
{
  char ch;

  printf ("Mit 'q' koennen Sie dieses Programm beenden!\n");
  do
  {
      if (scanf(" %c", &ch)==0)
      {
          printf ("Unzulaessige Eingabe!");
          return 0;
      }
      zaehler();
  }
  while(ch != 'q');
  return 0;

}/* main */
```

Listing 10.2: Zählerfunktion

10.1.4 Namen von Funktionen

Es ist zu empfehlen, Substantive, die das Ergebnis der Funktion bezeichnen, als Funktionsnamen zu wählen (z.B. `mittelwert`). Logische Funktionen wird man vorzugsweise mit Adjektiven bezeichnen (z.B. `kleiner`), Funktionen ohne Rückgabewert hingegen mit

einem Verb, das die Aktion der Funktion beschreibt (z.B. `halloausgeben`). Wichtig ist vor allem, dass der Name der Funktion beschreibt, was diese Funktion macht.

10.2 Wertübergabe

10.2.1 Rückgabewert

Eine Funktion liefert meist einen Wert zurück. In unserem Beispiel erfolgte der Datenaustausch zwischen Hauptprogramm und der Funktion über die globalen Variablen. Der berechnete Mittelwert steht in der globalen Variablen `mw` zur Verfügung. Diesen Mittelwert soll die Funktion aber als Funktionswert zurückliefern. Dazu benötigen wir die `return`-Anweisung:

```
return mw;
```

Wie der Name *return* schon aussagt, erfolgt durch die `return`-Anweisung der Rücksprung in das Hauptprogramm, gleichzeitig wird der angegebene Wert zurückgegeben. Die allgemeine Form dieser Anweisung ist

```
return Ausdruck ;
```

Unsere Funktion liefert nun einen Wert zurück. Dieser Wert kann dann im Hauptprogramm beim Aufruf der Funktion verwendet werden:

```
float arr[100];
int n=100;

float mittelwert( )
{
  float sum, mw;
  int i;
  sum=0.0;
  for (i=0; i<n; i++)
  sum=sum+arr[i];
  mw=sum/n;
  return mw; /* Rückgabewert */
}

int main()
{
  float x,y,z;
  int i,j,k;
  ...
```

```
  z = mittelwert(); /* Aufruf der Funktion */
  ...
}
```

Der Aufruf der Funktion `mittelwert()` liefert den Mittelwert des Feldes `arr`. Mit diesem Wert, vom Typ `float`, kann gerechnet werden; er kann zum Beispiel einer Variablen zugewiesen werden. Die Variable `float mw` ist jetzt nur noch eine lokale Variable, die im Hauptprogramm nicht sichtbar ist. Sie kann ganz entfallen, wenn mit `return sum/n;` der berechnete Mittelwert gleich zurückgegeben wird. In C ist es auch möglich, den Rückgabewert einer Funktion bei ihrem Aufruf zu ignorieren. Eine Warnung vom Compiler gibt es dabei nicht.

Im vorigen Beispiel haben wir die `return`-Anweisung verwendet. Das Symbol `return` gefolgt von dem Ausdruck, der das Ergebnis der Funktion darstellt, beendet die Ausführung der Funktion. Die `return`-Anweisung kann an beliebigen Stellen innerhalb der Funktion stehen, auch mehrfach, normalerweise aber steht sie am Schluss unmittelbar vor der geschweiften Klammer `}`.

Allgemein sieht die `return`-Anweisung so aus:

```
return -Anweisung ::= " return " [Ausdruck] " ; " .
```

Eine `return`-Anweisung ohne darauffolgenden Ausdruck beendet die Funktion, der Rückgabewert ist dann aber undefiniert. Eine Funktion, die keinen Rückgabewert liefert, kann auch sinnvoll sein, es muss jedoch immer ein Rückgabetyp angegeben werden: die Funktion ist dann vom Typ `void`.

Beispiel:

```
int z;

void qprint()
{
  printf("%d \n", z * z);
  return;
}
```

Selbst das Hauptprogramm (die Funktion `main()`) hat einen Rückgabetyp. Vielleicht haben Sie sich ja schon gewundert, warum wir immer `int main()` geschrieben haben. Dieser Wert wird dann an das Betriebssystem zurückgegeben: auch ein ganzes Programm hat also einen Rückgabewert. Wir haben diesen aber bislang einfach ignoriert (kein `return` in `main`), so dass der Rückgabewert unserer Programme auch unbestimmt war.

Häufig wird in UNIX der Rückgabewert eines Programms zur Erkennung und Unterscheidung von Fehlern benutzt: das Programm gibt quasi eine Fehlernummer zurück.

```
#include <stdio.h>                    /* nötig für Ein-/Ausgabe */

float arr[100];
int n;

float mittelwert ( )      /* Funktionskopf    */
{
  int i;
  float sum, mw;

  sum=0.0;              /* -                  */
  for (i=0; i<n; i++) /* | Funktionsblock */
    sum=sum+arr[i];   /* |                  */
  mw=sum/n;           /* -                  */

  return mw;
}


int main()                          /* Programm-Kopf */
{
  float z;
  int i;
  do
  {
    printf ("Bitte  Anzahl der Werte (<=100) eingeben! n = ");
    if (scanf("%d", &n)==0)
    {
        printf ("Unzulaessige Eingabe!");
        return 1;
    }
  }
  while(n > 100);

  printf ("Bitte  geben Sie nun die einzelnen Werte ein!\n");
  for(i = 0; i < n; i++)
  {
    printf ("Wert %i: ", i+1);
    if (scanf("%f", &arr[i])==0)
    {
        printf ("Unzulaessige Eingabe!");
        return 1;
    }
  }

  z = mittelwert ();
  printf ("\nEs ergibt sich ein Mittelwert von %f!\n\n", z);
  return 0;

}/* main */
```

Listing 10.3: Mittelwertfunktion mit Rückgabe

Wenn diese „Fehlernummer“ gleich Null ist, bedeutet dies, dass kein Fehler aufgetreten ist. Wenn Sie „saubere“ Programme schreiben wollen, sollte Ihre `main`-Funktion (im Erfolgsfall) immer 0 zurückgeben.

10.2.2 Parameterübergabe

Hier finden Sie zwei einfache Funktionen, die die Fakultät von `zahl` und die Fläche eines Kreises mit dem Radius `rad` berechnen:

```
int zahl;

int fakultaet() /* Berechnung der Fakultät von zahl */
{
  int i=1, x=1;
  while (i<=zahl)
  {
    x=x*i;
    i++;
  }
  return x;
}


double rad;

double flaeche()
{
  const double pi=3.14159;
  return rad * rad * pi;
}

int main()
{
  int fi;
  double hoehe=2.0, volumen;
  zahl = 25;
  rad = 12.345;

  fi = fakultaet();
  volumen = flaeche() * hoehe;
  ...
}
```

Beide Funktionen liefern zwar den gewünschten Wert, sie sind aber ziemlich umständlich zu benutzen: es wird nur die Fakultät der globalen Variablen `zahl`, bzw. die Kreisfläche für die globale Variable `rad` berechnet. Dies macht die allgemeine Verwendung ziemlich umständlich. Aber auch hierfür gibt es eine Lösung.

Wenn die Funktionen auf verschiedene Variablen verwendbar sein sollen, so muss die Datenübergabe mittels eines Parameters oder einer Parameterliste durchgeführt werden, anstatt über globale Variable.

Wie sieht das in unserem Beispiel aus? Betrachten wir zunächst die Definition der beiden Funktionen. In den runden Klammern stehen die Formalparameter, jeweils mit Typ. Ein Formalparameter ist der Name einer lokalen Variable, die den Wert des aktuellen Parameters annimmt: wir wollen uns ja von der Abhängigkeit von globalen Variablen befreien, und nur die Parameter als Wert an die Funktion übergeben (das ist dann der aktuelle Parameter). Dieser Wert muss aber in der Funktion auch irgendwie bekannt sein: genau das erledigen die formalen Parameter. Sie nehmen innerhalb der Funktion den Wert der aktuellen Parameter an.

```
int fakultaet(int n)          /* Berechnung der Fakultät von n */
{
  int i=1, x=1;
  while (i<=n)
  {
    x=x*i;
    i++;
  }
  return x;
}

double flaeche(double radius) /* Berechnung der Kreisfläche */
{
  const double pi=3.14159;
  return radius * radius * pi;
}
```

Der Aufruf der Funktion erfolgt dann durch die entsprechenden aktuellen Parameter, die an gleicher Stelle in den runden Klammern angegeben werden.

```
int main()
{
  int zahl=7, fi;
  double rad=4.5, hoehe=2.0, f, volumen;
  double a=8.1, b=8.5, c;
```

```
  fi = fakultaet(zahl);
  volumen = flaeche(rad) * hoehe;
  c = flaeche(a);

  volumen = flaeche(b) * hoehe;
  ...
}
```

Der Parameter `int n` bzw. `double radius` wurde in der Parameterliste im Funktionskopf eingeführt. `n` oder `radius` sind formale Parameter und stellen dabei einen Platzhalter für die aktuellen Werte dar, die beim Funktionsaufruf angegeben werden:

```
fi = fakultaet(zahl);
c = flaeche(a);
```

Es wird dabei immer der Wert der Variablen (hier `zahl` und `a`) übergeben, man könnte natürlich auch direkt eine Zahl angeben:

```
double f;
f = flaeche(2.5);
```

Funktionen dürfen innerhalb eines Ausdrucks aufgerufen werden. Der Name der Funktion steht dabei zugleich für das Ergebnis. Eine Parameterliste (Klammern) muss vorhanden sein, auch wenn sie leer ist. Mögliche Aufrufe sind daher:

```
u = fakultaet(10);
s[i] = fakultaet(x) + fakultaet(y);
s[i-1] = x * fakultaet(x);
```

Sie wollen einer Variablen `u` die Differenz aus 8! und 4! zuweisen. Wie sieht der entsprechende Ausdruck aus?

```
u = fakultaet(8) - fakultaet(4);
```

Beim Funktionsaufruf kann ein aktueller Parameter auch ein Ausdruck oder eine Konstante sein. Folgende Aufrufe sind also auch möglich:

```
u = fakultaet(12);
u = fakultaet(k+5);
u = fakultaet( 5*a + 7 )
```

```
#include <stdio.h>                    /* nötig für Ein-/Ausgabe */

int fakultaet (int n) /* Berechnung der Fakultät von n */
{
  int i=1, x=1;
  while (i<=n)
    {
      x=x*i;
      i++;
    }
  return x;
}

double flaeche (double radius) /* Berechnung der Kreisfläche */
{
  const double pi=3.14159;
  return radius * radius * pi;
}


int main()                            /* Programm-Kopf */
{
  int zahl=7, fi ;
  double hoehe=2.0, volumen;
  double a=8.1, b=8.5, c;
  fi  = fakultaet (zahl);
  c = flaeche(a);
  volumen = flaeche(b) * hoehe;
  printf ("Berechnete Werte: fi = %d volumen = %f c = %f\n", fi, volumen, c);
  c = fakultaet (12) + fakultaet (5 * zahl);
  printf ("fakultaet (12) + fakultaet (5 * zahl) mit zahl = 7 ergibt: %.2f\n", c);
  return 0;
}/* main */
```

Listing 10.4: Parameterübergabe
Das Programm zeigt einige Varianten der Parameterübergabe.

Ein weiteres Beispiel:

```
int quadrat(int z)
{
  return ... ;
}
```

Ergänzen Sie die fehlenden drei Zeichen, wenn mit dieser Funktion das Quadrat einer Zahl berechnet werden soll.

```
int quadrat(int z)
{
  return z*z;
}
```

Man muss bei der Parameterübergabe streng darauf achten, dass Anzahl und Reihenfolge der Parameter bei der Definition und beim Aufruf übereinstimmen. In ANSI-C erfolgt hierbei meist noch nicht mal eine Kontrolle durch den Compiler oder Linker. Wenn die Anzahl nicht übereinstimmt, kann dies dann zu undefinierten Effekten führen.

Eine Funktion ohne Eingabeparameter sollte daher mit einem `void` Parameter definiert werden.

```
void halloausgeben(void)
{
  printf("Hallo ! \n");
  return;
}
```

Der Aufruf erfolgt dann ohne Angabe aktueller Parameter:

```
halloausgeben();
```

Die Klammern sind hier aber dringend erforderlich.

Man kann aber nicht nur einen oder gar keinen Parameter angeben, es wurde ja bereits von der Parameterliste gesprochen. Hier werden die einzelnen Parameter einfach durch Komma getrennt nacheinander aufgeführt. Jeder Parameter muss mit Typ angegeben werden:

```
double multiply(double x, double y)
{
  return x*y;
}
```

10.2.3 Wertparameter (call by value)

Beim Aufruf der Funktion wird der formale Parameter aus der Definition

```
double flaeche(double radius)
```

durch den aktuellen Parameter ersetzt:

```
f = flaeche(rad);
```

In C gibt es nur die sogenannten Wertparameter (Wertaufruf = call by value). Das bedeutet, es wird nur der Wert des aktuellen Parameters an die Funktion übergeben, es wird eine Kopie des Wertes angelegt. Die Funktion hat damit keinen Zugriff auf die beim Aufruf angegebenen Variablen. In C ist also über die Parameterliste nur eine Eingabe in die Funktion möglich, eine Rückgabe von Werten über die Parameterliste ist nicht möglich.

Wir wollen nun eine Funktion schreiben, die aus dem Radius und der Höhe das Volumen eines Zylinders berechnet.

```
double zylinder(double radius, double h)
{
  double fl;
  fl = flaeche(radius);
  return fl * h;
}
```

Hier wird die bereits bekannte Funktion `flaeche` zur Berechnung der Grundfläche des Zylinders benutzt. Ein Funktionsaufruf kann dann so aussehen:

```
double radius, hoehe, volumen;
volumen = zylinder(radius, hoehe);
```

```
#include <stdio.h>                    /* nötig für Ein-/Ausgabe */

double flaeche (double rad)
{
  const double pi=3.14159;
  return rad * rad * pi;
}

double zylinder (double radius, double h)
{
  double fl;
  fl  = flaeche(radius);
  return fl  * h;
}

int  main()                           /* Programm-Kopf */
{
  double radius, hoehe, volumen;
  printf ("Bitte  den Radius des Zylinders eingeben! r = ");
  if (scanf("%lf",  &radius)==0)
  {
      printf ("Unzulaessige Eingabe!");
      return 1;
```

```
    }
    printf ("Bitte die Hoehe des Zylinders eingeben! h = ");
    if (scanf("%lf", &hoehe)==0)
    {
        printf ("Unzulaessige Eingabe!");
        return 1;
    }
    volumen = zylinder (radius, hoehe);
    printf ("Der Zylinder hat ein Volumen von %f!\n", volumen);
    return 0;
}/* main */
```

Listing 10.5: Volumenberechnung Zylinder

10.2.4 Nebeneffekte

Eine Funktion kann aber nicht nur auf ihre Parameter zugreifen, es sind ja auch alle globalen Variablen existent (und im allgemeinen auch sichtbar). Wertzuweisungen innerhalb einer Funktion zu nicht-lokalen (globalen) Variablen heißen auch Nebeneffekte (side effects). Sie sollten wegen ihrer unüberschaubaren Wirkungen aber vermieden werden.

Wenn Nebeneffekte unvermeidbar sind, oder es einen guten Grund für sie gibt, sollte man sie unbedingt gut dokumentieren (Kommentare!).

10.2.5 Mehrere Rückgabewerte

Bei der Definition einer Funktion sieht man schon, dass sie immer nur maximal einen Rückgabewert haben kann. Auch wird die Funktion beim Aufruf der `return`-Anweisung beendet. Was aber, wenn man eine Funktion mit mehreren Rückgabewerten schreiben will? Häufig ist das schon ein guter Hinweis, dass man doch besser für jeden der Werte eine eigene Funktion schreiben soll. Es gibt aber natürlich Fälle, in denen das nicht sinnvoll ist, wenn diese Rückgabewerte miteinander zusammenhängen oder gar nur zusammen berechnet werden können.

Ein Beispiel hierfür wäre eine Funktion, die ein Punkt im Raum mit seinen drei Koordinaten x, y und z berechnen soll. Man könnte nun wieder auf globale Variablen zurückgreifen, aber genau das sollte man in den meisten Fällen lieber vermeiden. Ein wesentlich elegantere Lösung ist hier die Verwendung von Strukturen (siehe Datentyp `struct`, 8.4). Der Typ unserer Struktur muss dann sowohl beim Aufruf der Funktion, als auch bei deren Verwendung bekannt sein: man definiert die Struktur also global, aber die Strukturvariablen dann nur lokal.

Sehen wir uns ein einfaches Beispiel an:

```
struct Punkt {int x, y, z;};

struct Punkt spiegelung(struct Punkt p)
```

```
{
  p.x = -p.x;
  p.y = -p.y;
  p.z = -p.z;

  return p;
}

int main()
{
  struct Punkt aktuell, neu;

  aktuell.x=5;
  aktuell.y=10;
  aktuell.z=-13;

  neu = spiegelung(aktuell);
}
```

Der Strukturtyp `Punkt` wird global vereinbart, so dass `main` und `spiegelung` beide diesen Typ kennen. In `main` werden die Variablen `aktuell` und `neu` verwendet, in `spiegelung` nur `p`. Wir haben also keine globalen Variablen.

In vielen Fällen wird man statt der Rückgabe einer Struktur aber auf Zeiger zurückgreifen (vgl. Kapitel 11.2.5).

10.2.6 Übergabe von Feldern

Kommen wir nochmals zu unserem Anfangsbeispiel, der Funktion `mittelwert`, zurück. Sie arbeitet mit den außerhalb vereinbarten, also globalen, Feldern und den Variablen `arr` und `n`:

```
float arr[100];
int n;

float mittelwert()
{
  float sum, mw;
  int i;
  sum=0.0;
  for (i=0; i<n; i++)
    sum=sum+arr[i];
  mw=sum/n;
```

```
  return mw;
}
```

Wenn die Funktion `mittelwert` auf verschiedene Arrays anwendbar sein soll, ist dieser Vorschlag noch ungünstig. Wir müssen also auch das Feld und seine Länge als Parameter übergeben.

```
float mittelwert(float arr[100], int n)
{
  float sum, mw;
  int i;
  sum=0.0;
  for (i=0; i<n; i++)
    sum=sum+arr[i];
  mw=sum/n;
  return mw;
}
```

Wie sieht jetzt der Funktionsaufruf innerhalb des Programms aus, wenn Sie den Mittelwert des Arrays `float feld[100]` berechnen wollen?

```
int main()
{
  float feld[100], a[50], b[500];
  float mwf, mwa, mwb;
  ...
  mwf = mittelwert (feld, 100);
  ...
}
```

Hier wird natürlich nicht das ganze Feld als Werteparameter (by value) an die Funktion übergeben. Felder können ja beliebig groß werden; wenn jedesmal der gesamte Inhalt eines riesigen Feldes mit übergeben würde, wären C-Programme häufig nur noch mit dem Kopieren von Daten beschäftigt. Man hat daher bei Feldern eine andere Lösung gefunden.

Wenn wir den Feldnamen ohne Index verwenden, so ist dies die Adresse des Feldes, wie wir dies schon bei den String-Arrays kennengelernt haben. Es wird also die Adresse des Feldes als Wertparameter an die Funktion übergeben. Genau diese Adresse erwartet aber auch die Funktion, wenn im Funktionskopf ein Feld als Formalparameter angegeben wird:

```
float mittelwert(float arr[100], int n)
```

Über diese Adresse kann nun die Funktion auf das Feld zugreifen und die einzelnen Elemente ansprechen:

```
sum = sum + arr[i];
```

Die Funktion arbeitet direkt mit dem Feld im Hauptprogramm, kann also auch die Feldelemente verändern. In der Tat haben wir hier also eher einen Referenzaufruf (call by reference). Nur die Adresse des Feldes wird „by value“ an die Funktion übergeben.

Achten Sie auch darauf, dass hier wirklich nur die Startadresse des Feldes übergeben wird, keine weitere Information über die Länge des Feldes, wie dies in anderen Programmiersprachen der Fall ist. Der Benutzer ist also für die Einhaltung der Indexgrenzen selbst verantwortlich. Daher geben wir ja auch die Länge des Feldes bei der Funktion extra mit an.

Im Funktionskopf haben wir als Formalparameter `float arr[100]` angegeben, in Wirklichkeit erwartet die Funktion aber nur die Adresse des Feldes. Die angegebene Dimension 100 ist völlig uninteressant, da ja für den Formalparameter kein Platz angelegt wird. Die Angabe der Dimension kann daher auch entfallen:

```
float mittelwert(float arr[], int n)
```

Die eckigen Klammern müssen aber bleiben, da es sich um ein Feld handelt, und nicht um eine einfache Variable.

Wie sieht der Aufruf aus, wenn jeweils Mittelwert der Arrays `a` und `b` berechnet werden soll? Die Adresse der Arrays `a` oder `b` ist dabei der aktuelle Parameter.

```
mwa = mittelwert(a, 50);
mwb = mittelwert(b, 500);
```

```
#include <stdio.h>                    /* nötig für Ein-/Ausgabe */

float mittelwert(float arr[100], int n)     /* Funktionskopf    */
{
  int i;
  float sum, mw;

  sum=0.0;              /* -                    */
  for (i=0; i<n; i++)   /* | Funktionsblock     */
    sum=sum+arr[i];     /* |                    */
  mw=sum/n;             /* -                    */
  return mw;
}
```

```
int main()                          /* Programm-Kopf */
{
  float z, feld[100];
  int i, n;

  do
  {
    printf ("Bitte Anzahl der Werte (<=100) eingeben! n = ");
    if (scanf("%d", &n)==0)
    {
        printf ("Unzulaessige Eingabe!");
        return 1;
    }
  }
  while(n > 100);

  printf ("Bitte geben Sie nun die einzelnen Werte ein!\n");
  for(i = 0; i < n; i++)
  {
    printf ("Wert %i: ", i+1);
    if (scanf("%f", &feld[i])==0)
    {
        printf ("Unzulaessige Eingabe!");
        return 1;
    }
  }/* for */

  z = mittelwert(feld, n);
  printf ("\nEs ergibt sich ein Mittelwert von %f!\n\n", z);
  return 0;

}/* main */
```

Listing 10.6: Übergabe von Feldern

10.2.7 Referenzparameter (call by reference)

In vielen Programmiersprachen wird zwischen Wertparametern (call by value) und Referenzparametern (call by reference) unterschieden. Reines ANSI-C kennt keine Referenzparameter; hier muss man sich mit Zeigern behelfen (siehe auch: Kapitel 11.2.5).

Bei Wertparametern wird nur der Wert übergeben (kopiert), eine Änderung des Wertes in der Funktion hat keine Auswirkung auf die Variable, die beim Funktionsaufruf als Parameter übergeben wurde.

Für die Verwendung in Funktionen bedeutet das für die beiden Arten von Parametern:

- Wertparameter

Wenn sich der formale Wertparameter in der Funktion ändert, ändert sich dadurch nicht der aktuelle Wertparameter. Man kann daher einen Wert auch in Form eines Ausdrucks oder durch eine Konstante in die Funktion einbringen.

- Felder
  Ganze Felder werden nicht als Wertparameter übergeben, sondern es wir nur die Adresse des Arrays als Wertparameter übergeben. Damit hat die Funktion Zugriff auf das beim Aufruf angegebene Feld und kann dieses lesen und verändern. Für den Programmierer ist dies ein Referenzparameter (call by reference).

10.3 Deklaration von Funktionen

Wir haben bisher kennengelernt, wie Funktionen definiert werden und wie sie aufgerufen werden. In C werden Funktionen immer außerhalb des Hauptprogrammes definiert, jede Funktion wird getrennt definiert, eine Schachtelung (lokale Funktionen) ist in C nicht möglich.

Wir sind davon ausgegangen, dass die Funktion beim Aufruf im Programm bekannt ist, dass dem Compiler bekannt ist, welchen Datentyp die Funktion zurückliefert und welche Parameter sie erwartet. Dies ist dadurch gewährleistet, dass die Definition der Funktion in der gleichen Datei vor der Verwendung erfolgte.

Da das Hauptprogramm main ebenso wie die Funktionen jeweils separate Programmteile sind, ist diese Reihenfolge nicht unbedingt erforderlich. Wenn die Definition der Funktion nach dem Hauptprogramm oder in einer eigenen Datei steht, muss die Funktion aber vorher deklariert werden, damit sie richtig verwendet werden kann. Die Deklaration einer Funktion muss vor ihrer ersten Verwendung geschehen.

Was unterscheidet eine Deklaration von einer Definition? Die Definition einer Funktion ist die Vereinbarung des Namens und Typs der Funktion, der Parameter und des Funktionsblocks (Implementierung). Bei der Deklaration werden nur Name, Typ und Parameter angegeben, die gesamte Implementierung erfolgt an einer anderen Stelle. Die Deklaration enthält also alles, was man zur Benutzung der Funktion wissen muss. Speziell der Compiler benötigt diese Informationen, um den Funktionsaufruf richtig übersetzen zu können.

Man könnte nun immer einfach alle Funktionen in der „richtigen“ Reihenfolge schreiben, dann wäre eine Deklaration eigentlich überflüssig. Nur müsste sich dann der Programmierer auch noch um die Reihenfolge der Funktionen sorgen. Wenn sich zwei Funktionen gegenseitig aufrufen sollen (kann durchaus sinnvoll sein!), geht das sogar nur mit einer Deklaration der Funktionen.

Was muss genau in der Deklaration stehen? Alle erforderlichen Informationen finden wir bereits im Funktionskopf. Die Deklaration einer Funktion sieht eigentlich genau

so aus wie der Funktionskopf, nur folgt statt des Funktionsblocks direkt ein Strichpunkt. Dabei dürfen die Namen der Variablen in der Parameterliste auch wegelassen werden, nur deren Typ muss angegeben werden.

In unseren Beispielen wäre eine Deklaration so möglich:

```
/* Deklaration der Funktionen: */

int fakultaet(int n);      /* "n" könnte man hier auch weglassen */
double flaeche(double);    /* hier wurde "radius" weggelassen    */

/* Hauptprogramm: */

int main()
{
  int zahl;
  double rad, a, b, c;
  int fi;
  double hoehe=2.0, volumen;

/* ... */

fi = fakultaet(zahl);                /* Aufruf */
volumen = flaeche(rad) * hoehe;
c = flaeche(a);

/* ... */
}

/* Berechnung der Fakultät von n */

int fakultaet(int n)  /* Definition: "n" hier NICHT weglassen!*/
{
  int i=1, x=1;
  while (i<=n)
  {
    x=x*i;
    i++;
  }
  return x;
}

/* Berechnung der Kreisfläche */
```

```
double flaeche (double radius)          /* Definition */
{
const double pi=3.14159;
return radius * radius * pi;
}
```

Die Deklaration von Funktionen könnte, wie bei Variablen, auch innerhalb eines Hauptprogrammes oder einer Funktion erfolgen. Sie kann aber auch außerhalb erfolgen, dann gilt sie global ebenso für alle folgenden Programmteile.

Diese Funktionsdeklaration beinhaltet alle notwendigen Informationen für den Compiler. Sie gibt Auskunft über den Typ der Funktion, also den Typ des Return-Wertes. Damit kann der Compiler das Bitmuster richtig interpretieren und ggf. eine Typwandlung vornehmen. Fehlt die Deklaration, so wird als Normalfall der Datentyp `int` angenommen, damit wird der Returnwert falsch interpretiert und verwendet.

In der Funktionsdeklaration steht auch der Typ der Formalparameter. Wird beim Aufruf der Funktion als aktueller Parameter ein anderer Datentyp verwendet, so wird der aktuelle Wert in den passenden Datentyp gewandelt, entsprechend der Funktionsdeklaration. Wird z.B. die Funktion

```
double flaeche(double radius);
```

mit einem Integer Wert aufgerufen:

```
x = flaeche(10);
```

so erfolgt eine Typwandlung: 10 $\Rightarrow$ 10.0, also der richtige Aufruf

```
x = flaeche (10.0);
```

Ohne eine Deklaration der Funktion kann diese Wandlung nicht erfolgen. Je nach Compiler wird dann falsch gerechnet, oder das Programm kann nicht übersetzt werden. Man sollte also am besten alle Funktionen (ausgenommen das Hauptprogramm `main`) deklarieren, auch wenn das durch ihre Reihenfolge gar nicht notwendig wäre. Schaden kann es nicht.

10.4 Rekursion

In C ist es erlaubt, wie auch in den meisten anderen Programmiersprachen, dass sich eine Funktion selbst aufruft. Ist dies der Fall, so spricht man von Rekursion. Zum Beispiel ist die Funktion `rek` rekursiv:

```
int rek(int n)
{
  if (n == 0)
    return 0;
  else
    return n + rek(n - 1);    /* hier rekursiver Aufruf! */
}
```

Die Funktion `rek(n)` berechnet die Summe aller Zahlen von n bis 0.

Ähnlich ist die Fakultät einer natürlichen Zahl N, also FAK(N) oder einfach N!, definiert als:

```
FAK(N) = N · (N-1) · (N-2) · ... · 1 (falls N > 0)
FAK(0) = 1
```

Ebenso gilt:

```
FAK(N) = N · FAK(N-1) (falls N > 0)
```

Diese Definition lässt sich einfach in eine C-Funktion übertragen:

```
int fak (int n)
{
  if (n == 0)
    return 1;               /* da FAK(0) = 1 */
  else
    return n * fak(n-1);    /* hier Rekursion */
}
```

Eine andere rekursive Funktion ist die Fibonacci-Funktion. Sie ist definiert als:

```
FIB(N) = FIB(N-1) + FIB(N-2) (falls N > 1)
FIB(0) = 1
FIB(1) = 1
```

Das dazu passende C-Programm ist damit:

```
int fib(int n)
{
  if (n == 0 || n == 1)
    return 1;
  else
    return fib(n-1) + fib(n-2);
}
```

Als letztes Beispiel zur Rekursion sei die Berechnung des größten gemeinsamen Teilers zweier Zahlen a und b angeführt. Der GGT könnte iterativ, also mit einer Schleife, berechnet werden. Dabei sind drei Fälle zu unterscheiden:

I. ggt(a, b) = a (falls a = b)

II. ggt(a, b) = ggt(a-b, b) (falls a > b)

III. ggt(a, b) = ggt(a, b-a) (falls b > a)

Diese Fallunterscheidung lässt sich aber auch sehr einfach durch eine rekursive Funktion realisieren. Man erhält folgende C-Funktion:

```
int ggt(int a, int b)
{
  if (a == b)
    return a;
  else if (a > b)
    return ggt(a-b, b);
  else
    return ggt(a, b-a);
}
```

```
#include <stdio.h>              /* nötig für Ein−/Ausgabe */

int fak(int n)
{
  if (n == 0)
    return 1;                   /* da FAK(0) = 1 */
  else
    return n * fak(n−1);        /* hier Rekursion */
}/* fak */

int fib (int n)
{
  if (n == 0 || n == 1)
    return 1;
  else
    return fib (n−1) + fib(n−2);
}/* fib */

int ggt(int a, int b)
{
  if (a == b)
    return a;
```

```
  else if (a > b)
    return ggt(a−b, b);
  else
    return ggt(a, b−a);
}/∗ggt∗/

int main()                    /∗ Programm−Kopf ∗/
{
  int i, j;
  printf ("Von welcher Zahl wollen Sie die Fakultaet berechnen lassen? ");
  if (scanf("%d", &i)==0)
  {
      printf ("Unzulaessige Eingabe!");
      return 1;
  }
  printf ("Ergebnis: %i\n", fak(i));
  printf ("Fuer welches N wollen Sie die Fibonacci−Zahl berechnen lassen? N = ");
  if (scanf("%d", &i)==0)
  {
      printf ("Unzulaessige Eingabe!");
      return 1;
  }
  printf ("Ergebnis: %i\n", fib (i));
  printf ("Der groesste gemeinsame Teiler soll berechnet werden.\n");
  printf ("Bitte Zahl 1 eingeben:");
  if (scanf("%d", &i)==0)
  {
      printf ("Unzulaessige Eingabe!");
      return 1;
  }
  printf ("Bitte Zahl 2 eingeben: ");
  if (scanf("%d", &j)==0)
  {
      printf ("Unzulaessige Eingabe!");
      return 1;
  }
  printf ("Ergebnis: %i\n", ggt(i, j));
  return 0;
}/∗main∗/
```

Listing 10.7: Rekursion

10.5 Vertiefung: Funktionen und Variablen

Der Zusammenhang von Funktionen, lokalen und globalen Variablen, Wert- und Referenzparametern sowie die Rekursion bereiten häufig Verständnisprobleme. In diesem

Vertiefungskapitel sollen daher diese Problemfelder nochmal wiederholt und etwas anders dargestellt werden.

In der Hoffnung und Überzeugung, dass ein grobes Verständnis für die inneren Abläufe auch bei der Anwendung hilft, wird hier anhand von Beispielen ein Blick „unter die Haube“ geworfen, z.B. um zu sehen, wie Variablen im Speicher abgelegt werden. Vorher aber noch ein paar Worte zu einer häufig gestellten Frage:

10.5.1 Wie teilt man ein Programm in Funktionen auf?

Diese Frage lässt sich, wie so viel Fragen beim Softwareentwurf, nicht allgemein gültig beantworten. Man kann aber einige Tipps geben, wie die Funktionalität eines Programms in einzelne Funktionen aufgeteilt werden kann.

Immer wiederkehrende Aufgaben sind noch ein relativ offensichtlicher Kandidat für eine Funktion. Wenn Sie also in einem Programm mehrere Zeilen kopieren, ist das schon mal ein Hinweis, dass Sie diesen Abschnitt besser als eine Funktion implementieren sollten. Der Vorteil ist vor allem darin zu sehen, dass Sie dann mögliche Fehler nur an einer Stelle korrigieren müssen. Durch diese Wiederverwendung sparen Sie sich auch Zeit beim Schreiben und machen das Programm kürzer und übersichtlicher.

Wenn einzelne Schritte nacheinander durchzuführen sind, dann ist es auch häufig besser, die Einzelschritte in eigene Funktionen zu schreiben. Auch wenn diese Funktionen evtl. nur an einer Stelle verwendet werden, sieht man (bei vernünftiger Namengebung) viel schneller, welche Schritte nacheinander ausgeführt werden, als wenn sich diese über viele Zeilen Code erstrecken („Spaghetti-Code“).

Ein weiteres Kriterien für das Aufteilen in Funktionen ist die Anzahl an Parametern, die übergeben werden. Eine ellenlange Parameterliste macht Programme nur unübersichtlich. Einerseits kann man dann oft mit Strukturen abhelfen, indem man logisch zusammengehörende Parameter auch in einer Struktur zusammenfasst, oder man findet besser geeignete Funktionen.

Ein ganz pauschales Kriterium ist schließlich die Länge der Funktionen (auch die der `main`-Funktion). Man kann zwar keine genauen Grenzen angeben, das hängt immer vom Einzelfall ab. Wenn aber eine Funktion so lange wird, dass sie sich über mehrere Bildschirmseiten erstreckt, ist diese Grenze ziemlich sicher überschritten, und es wird höchste Zeit für eine Aufteilung.

Nach diesen allgemeinen Bemerkungen aber wieder zurück zu den „Innereien“ von Funktionen und Variablen.

10.5.2 Variablen (allgemein)

Sehen wir uns einmal an, was ein Compiler aus einer normalen Variable macht. Nehmen wir die „kleinste“ Variable (die mit dem kleinsten Wertebereich), ein `char`:

```
char mych = 'A';
```

Der Computer muss sich diese Variable nun irgendwie merken (speichern). Dazu steht ihm sein Speicher zur Verfügung, der aus einer Menge einzelner Zellen aufgebaut ist. Jede dieser Speicherzellen hat nun ihre eigene Nummer (Adresse), über die sie angesprochen (adressiert) werden kann. In Abb. 10.2 ist ein Ausschnitt aus dem Speicher dargestellt: links die Adresse, rechts der Inhalt (Fragezeichen, wenn wir den Inhalt dieser Zellen nicht kennen).

Abbildung 10.2: Adresse einer Variable im Speicher

So eine Speicherzelle kann man sich wie ein Postfach vorstellen, das ja auch seine Postfachnummer hat.

Unsere Variable `mych` muss also irgendwo im Speicher abgelegt werden. Wo genau die Variable liegt, ist hier eigentlich nicht weiter interessant, aber nehmen wir mal an, es ist an der Adresse 500 (rechts im Bild). Ein `char` begnügt sich gewöhnlich mit einer einzigen Speicherzelle.

Man könnte sich zwar auch vorstellen, dass der Programmierer die einzelnen Speicherzellen immer direkt über ihre Adresse anspricht, was aber natürlich enorm unübersichtlich und wenig komfortabel wäre. Eine Variable ist also im Grunde nur ein Pseudonym, also ein anderer Name für eine bestimmte Speicherzelle. Zusätzlich wird der Programmierer von der Aufgabe befreit, sich eine Speicherzelle auszusuchen. Beide Aufgaben nimmt ihm der Compiler ab: beim Übersetzen des Programms legt er fest, welche Variable wo gespeichert wird, und statt des Namens der Variablen wird dann im Maschinenprogramm die Adresse eingesetzt.

10.5.3 Lokale Variablen

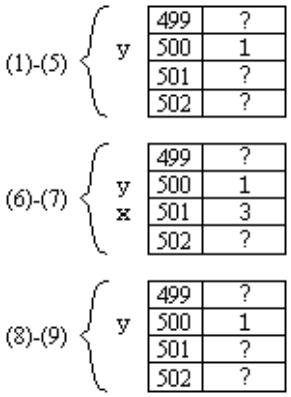
Wir haben gesehen, dass Variablen nur eine andere Bezeichnung für eine bestimmte Speicherzelle sind. Wie passt das aber mit lokalen Variablen zusammen? Eigentlich ganz gut, nur gilt dieser Zusammenhang nicht während der Ausführung des ganzen Programms, sondern nur solange die jeweilige Funktion ausgeführt wird.

Existenz

Bei jedem Aufruf einer Funktion werden für alle automatischen lokalen Variablen neue Speicherzellen reserviert, die dann über den Namen der Variablen angesprochen werden. Nach dem Ende der Funktion wird dieser Speicher wieder freigegeben. In der Zeit zwischen Anlegen und Freigeben einer Variable am Beginn bzw. Ende einer Funktion ist diese Variable existent; vorher und nachher existiert sie nicht.

Sehen wir uns folgendes Beispiel an:

```
#include <stdio.h>
(1) int y=1;
(5) void myfunc()
    {
(6)   int x=3;
(7)   printf("%d \n", x );
(8) }

(2) int main()
    {
(3)   printf("%d \n", y );
(4)   myfunc();
(9)   printf("%d \n", y );
    }
```



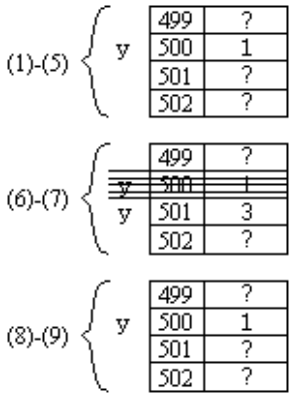
Vor einigen Zeilen ist eine Nummer angegeben, die den Ablauf des Programms nachvollzieht. Schritt (1) beinhaltet die Vereinbarung der globalen Variable `y`; diese ist während des gesamten Programmablaufs existent. Sie wird mit dem Startwert 1 initialisiert. In (2) wird dann das Hauptprogramm ausgeführt: hier wird (3) `y` ausgegeben und dann (4) `myfunc()` aufgerufen. Sobald (5) die Funktion `myfunc()` betreten wird, wird auch (6) die Variable `x` erzeugt, d.h. sie wird z.B. an der Adresse 501 im Speicher gehalten. Wegen der Initialisierung hat sie den Wert 3. Nun (7) wird `x` ausgegeben. Am Ende (8) von `myfunc()` wird `x` wieder freigegeben, die Speicherzelle 501 kann jetzt wieder für andere Aufgaben verwendet werden. Schließlich (9) wird nochmal `y` ausgegeben, dann ist auch `main()` fertig und unser Programm beendet sich.

Sichtbarkeit

Wir haben auch gesehen, dass eine globale Variable während des gesamten Programmablaufs existent ist. In den meisten Funktionen kann man auch auf sie zugreifen, sie ist dort sichtbar. Kann dann eine Variable auch unsichtbar werden? Nicht direkt, aber sie kann durch eine andere Variable verdeckt werden. Wenn in einer Funktion eine lokale Variable mit dem gleichen Namen wie eine globale Variable vereinbart wird, dann verdeckt diese einfach die globale Variable. Damit ist die lokale Variable existent und sichtbar, und die globale zwar existent (es gibt sie ja immer noch), aber nicht mehr sichtbar.

Sehen wir uns wieder ein Beispiel an:

```
#include <stdio.h>
(1) int y=1;
(5) void myfunc()
    {
(6)   int y=3;
(7)   printf("%d \n", y );
(8) }

(2) int main()
    {
(3)   printf("%d \n", y );
(4)   myfunc();
(9)   printf("%d \n", y );
    }
```



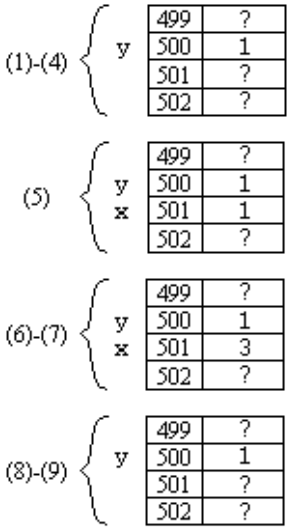
Dieses Beispiel ist fast identisch zum vorausgegangenen, in `myfunc()` wurde lediglich `x` durch `y` ersetzt; somit hat nun die lokale Variable in `myfunc()` auch den Namen |y|. In (5), also beim Betreten der Funktion `myfunc()` wird nun (6) die Variable `y` erzeugt, d.h. sie wird z.B. an der Adresse 501 im Speicher gehalten. Das globale `y` existiert weiter, und behält auch seinen Wert, nur ist mit `y` innerhalb von `myfunc()` die Adresse 501 gemeint, nicht mehr die 500. Am Ende (8) von `myfunc()` wird `y` wieder freigegeben, die Speicherzelle 501 kann jetzt wieder für andere Aufgaben verwendet werden. In (9) wird wieder das globale `y` ausgegeben, der Wert hat sich nicht geändert.

Wertparameter

Bei normalen Wertparametern handelt es sich im Grunde um lokale Variablen, denen beim Funktionsaufruf der Wert der aktuellen Parameter zugewiesen wird.

Unser Beispiel muss nur leicht geändert werden:

```
#include <stdio.h>
(1) int y=1;
(5) void myfunc(int x)
          {
(6)   int x=3;
(7)   printf("%d \n", x );
(8) }

(2) int main()
    {
(3)   printf("%d \n", y );
(4)   myfunc(y);
(9)   printf("%d \n", y );
    }
```



Dieses Beispiel ist fast identisch zur ersten Version, nur wird in `myfunc()` statt einer lokalen Variable `x` jetzt ein Wertparameter `x` benutzt. Als Wert wird `y` übergeben.

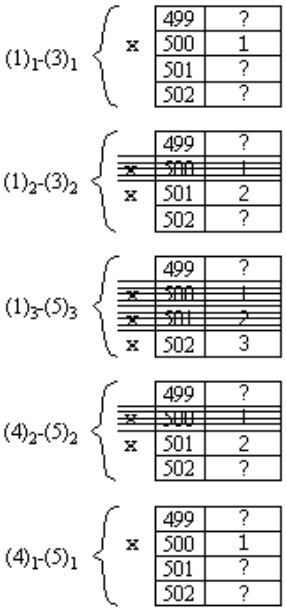
Wir sehen in (5), dass `x` auch hier eine neue Speicherzelle 501 zugewiesen wird. Diese Speicherzelle wird mit dem Wert des aktuellen Parameters, also 1 vorbelegt. In (6) wird der Wert von `x` auf 3 geändert und in (7) ausgegeben.

Rekursion

Eine automatische lokale Variable wird erst beim Funktionseintritt (also unmittelbar nachdem die Funktion aufgerufen wurde) erzeugt. Was passiert nun, wenn sich diese Funktion selbst aufruft? Auch nichts Anderes, es wird wieder eine neue Speicherzelle reserviert, auf die sich dann unsere lokale Variable bezieht. Die alte Speicherzelle bleibt natürlich reserviert, da die aufrufende Funktion noch nicht zu Ende ist und noch auf das Ende der gerufenen Funktion wartet.

Hier ist nun ein neues Beispiel an der Reihe:

```
#include <stdio.h>

(1) void myfunc(int x)
    {
(2)   if(x<3)
(3)   myfunc(x+1);
(4)   printf("%d \n", x );
(5) }

int main()
{
  myfunc(1);
}
```



Wir schreiben nun zu den einzelnen Schritten (1) bis (5) zusätzlich die Nummer des Durchlaufs; (7) 9 wäre der neunte Aufruf der Funktion `myfunc()`, aktuell an der Stelle 7.

Hier wird myfunc() zunächst mit dem Startwert 1 aufgerufen. Im ersten Durchlauf ist (1) 1 der Parameter x gleich 1. Wenn (2) `x` kleiner als 3 ist, wird `myfunc()` mit `x+1` aufgerufen, also jetzt der Wert 2. Nun wird in (1) 2 wieder eine neue Variable `x` erzeugt, die das erste `x` verdeckt. Es erfolgt nochmal ein Aufruf mit dem Wert 3, und es wird wieder (1) 3 ein neues `x` angelegt. Jetzt ist die `if`-Bedingung nicht mehr erfüllt, und (4) 3 das aktuelle `x` wird ausgegeben, also der Wert 3. Nun (5) 3 ist `myfunc()` fertig und kehrt zurück - zu `myfunc()`, an Stelle (4) 2. Von der Zeile davor wurde es ja im zweiten Durchlauf aufgerufen. Also wird das dort aktuelle `x` ausgegeben (Wert 2) und der (5) 2 zweite Durchlauf ist beendet. Die Funktion kehrt ebenfalls zurück, wieder an Stelle (4) 1 . Die Ausgabe ist nun 1, und die Funktion kann noch einmal zurückkehren. Da dies die oberste Ebene der Rekursion war, ist `func()` nun wirklich beendet.

Wir haben also bei jedem Aufruf eine neue Variable, auch wenn sie in der gleichen Funktion steht und den gleichen Namen trägt.

Statische Variablen

Was aber, wenn man das gar nicht will? Wenn eine Variable immer die selbe Speicherzelle bezeichnen soll, egal ob eine Funktion zum ersten oder hundertsten Mal aufgerufen wird, oder ob sich die Funktion gerade selbst aufruft? Man könnte natürlich globale Variablen nehmen, aber die sind dann im ganzen Programm sichtbar. Viel eleganter sind `static`-Variablen. Diesen wird, wie globalen Variablen auch, vom System beim Programmstart einmal eine feste Speicherzelle zugewiesen. Die Variable bezeichnet damit

immer die gleiche Speicherzelle, genau was wir erreichen wollten. Dennoch ist sie nur innerhalb der Funktion sichtbar, in der sie auch vereinbart wurde. Keine andere Funktion kann Unfug mit ihr treiben (außer die Variable wurde als Parameter an eine andere Funktion übergeben).

10.6 Bibliotheksfunktionen

In vielen Programmiersprachen sind häufig verwendete Standardfunktionen bereits in der Sprache selbst enthalten. Der Benutzer braucht sich nicht um ihre Vereinbarung zu kümmern, sondern kann sie nach Belieben aufrufen.

In C existieren in der Sprachdefinition selbst keinerlei Funktionen. Auch für viele Leistungen (wie z.B. Ein- und Ausgabeoperationen) sind keine eigenen Sprachelemente vorgesehen. Alle diese benötigten Funktionen sind in einer Standardbibliothek enthalten. Nach ANSI C ist der Inhalt dieser Bibliotheken festgelegt. Sie stellen daher eine wichtige Ergänzung der Sprache dar.

Eine Funktion, die in einem Programm verwendet wird, muss vor dem Aufruf deklariert werden, damit dem Compiler bekannt ist, welche Datentypen für den Rückgabewert und für die Parameter verwendet werden. Dies gilt für selbstgeschriebene Funktionen und ebenso für die Bibliotheksfunktionen. Die Deklarationen der Bibliotheksfunktionen sind in den zugehörigen Headerdateien enthalten. Diese Headerdateien müssen daher in das Programm eingefügt werden.

Ein Beispiel sind die Funktionen der Mathe-Bibliothek:

```
#include <math.h> /* Einfügungen */

int main()
{
  double x, y=0.5;
  x = sin(y);
}
```

Um die Funktion `sin()` zu benutzen, wird sie deklariert. Die Deklarationen der in der Bibliothek vorhandenen mathematischen Funktionen stehen in dem zugehörigen Headerfile *math.h*. Mit der Präprozessor-Anweisung

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

wird diese im System vorhandene Datei auch tatsächlich in das Programm eingefügt. Der Präprozessor wird automatisch vor dem Übersetzen des Programms aufgerufen. In der Datei *math.h* stehen u.a. die Zeilen (mehr zu dem Schlüsselwort `extern` siehe Kapitel 12.5)

```
extern double cos(double);
extern double sin(double);
extern double tan(double);
extern double exp(double);
extern double log(double);
extern double pow(double, double);
```

Die Funktion `sin` ist vom Typ `double` und liefert also einen Gleitkommawert zurück. Als Parameter wird auch ein `double` angegeben. In unserer Ausgabeanweisung `x = sin(y);` wird die Funktion aufgerufen und der Rückgabewert der Variablen `x` zugeweisen. Die Funktion `pow` hat zwei Parameter vom Typ `double`.

Strings sind in C kein eigener Datentyp, sondern werden in einem `char`-Feld abgelegt. Zur Verarbeitung von Strings sind einige Bibliotheksfunktionen vorhanden, zur Verwendung dieser Funktionen muss die entsprechende Headerdatei mit

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

eingefügt werden. Damit können z.B. die folgenden Funktionen verwendet werden:

```
extern char *strcpy(char *, const char *);
extern int strcmp(const char *, const char *);
```

In den Headerfiles stehen nur die Deklarationen der Funktionen, nicht der Funktionscode. Diese Information benötigt der Compiler, um den Aufruf der Funktionen richtig zu übersetzen, also die übergebenen Parameter und den Rückgabewert entsprechend zu verwenden. Die eigentliche Funktion wird dann vom Linker in der Regel automatisch hinzugefügt.

Warnung: Fehlende Deklarationen werden nicht als Fehler gemeldet. Das Programm arbeitet dann aber häufig fehlerhaft, da die Funktionen nicht ordnungsgemäß der Definition entsprechend verwendet werden.

Sie könnten sich die Datei *math.h* sogar ansehen, sie findet sich im Verzeichnis */usr/include/*. Allerdings ist dieser Header für Menschen kaum zu lesen: zum einen werden von dieser Datei wieder alle möglichen anderen Header eingebunden, außerdem werden einige „Tricks“ angewendet, z.B. um möglichst schnellen Code erzeugen zu können. Lassen Sie sich davon nicht verwirren - sie müssen den Inhalt dieses Headers ja nicht lesen oder gar verstehen.

```
#include <stdio.h>    /* nötig für Ein-/Ausgabe */
#include <string.h>   /*Einbinden der String-Funktionen*/

int main()            /* Programm-Kopf */
{
  int i, n;
  char string1[100], string2[100];
```

```
    string2[0] = 0;  /* string2 wird initialisiert  */
    printf ("Geben Sie einen String ein: ");
    if ( scanf("%s", string1))
    {
        printf ("Unzulaessige Eingabe!");
        return 1;
    }


    printf ("Der String hat eine Laenge von %d Zeichen.\n", (unsigned)strlen(string1));
    n = 100 / strlen (string1); /* wie oft passt string1 in string2 ? */

    for(i=0; i < n; i++)
    {
      strcat (string2, string1);
    }/* for */

    printf ("String2 enthaelt %i Kopien von string1:\n%s\n", n, string2);
    return 0;
}/* main*/
```

Listing 10.8: Bibliotheksfunktionen
Zwei Bibliotheksfunktionen aus *string.h* werden in diesem Programm verwendet. Die Funktionen sind `strlen` und `stringcat`.

10.7 Präprozessor

Vor der eigentlichen Übersetzung des C Programmes wird der Präprozessor aufgerufen. Mit den Präprozessor-Direktiven wird der Quellcode modifiziert, bevor er an den eigentlichen Compiler weitergegeben wird. Eine Präprozessor-Direktive kennen wir schon :

```
#include <filename>
```

Hier wird eine Datei in den Quellcode eingefügt, üblicherweise eine Headerfile:

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

Wozu werden diese Headerfiles benötigt, wozu werden sie in den Quellcode eingefügt? In den Headerfiles stehen die Deklarationen der Funktionen, die in der Bibliothek vorhanden sind. Hier steht also die Information, wie die Funktion zu benutzen ist, der Typ der Funktion, also des `return`-Wertes und der Typ der Parameter. Diese Information

benötigt der Compiler, die Headerfile sollte also am Anfang eingefügt werden. Die Headerfile enthält keinen Programmcode, wird also nicht vom Compiler übersetzt, sondern liefert nur die zur richtigen Übersetzung der Funktionen benötigte Information. Der eigentliche Programmcode ist als Objektmodul in der Bibliothek vorhanden und wird dann vom Linker dazugebunden (siehe auch Kapitel 12.1).

Die Headerfiles für die Bibliotheksfunktionen sind im System vorhanden, üblicherweise unter dem Pfad */usr/include/*. Auch selbstgeschriebene Funktionen müssen deklariert werden, also sollte man auch hierfür eine Headerfile erstellen, die dann in das Programm eingefügt wird. Diese Dateien findet man nicht unter dem allgemenen Sytem-Pfad. Um sie einzufügen verwendet man die Variante der Include-Anweisung

```
#include "filename"
```

z.B.

```
#include "myheader.h"
```

Dabei kann immer auch ein Pfad angegeben werden, wo die Datei zu finden ist, so könnten die oben genannten Headerfiles auch mit

```
#include "/usr/include/math.h"
```

eingefügt werden.

Allgemein könnte mit der `include`-Anweisung auch beliebiger Quellcode eingefügt werden. Dies ist aber nicht Sinn dieser Anweisung. Eine Vervielfältigung von Quellcode bläht natürlich nur das Programm auf.

Eine weitere interessante Präprozessor-Anweisung ist:

```
#define <identifier> <replacement>
```

Hier wird ein ganzer Name durch einen neuen Text ersetzt. Es handelt sich also, ähnlich wie bei einem Editor, um eine Textersetzung, die vor dem Übersetzen erfolgen soll. Sinnvoll ist dies, um einen Namen, der in verschiedenen Programmteilen vorkommt, an einer Stelle einheitlich durch einen Wert zu ersetzen, z.B.:

```
#define SCHRANKE 16.0
```

Damit wird an allen Stellen im Programm der Name `SCHRANKE` durch `16.0` ersetzt, danach wird das Programn übersetzt.

Ein weiteres Beispiel:

```
#define DIM 100
int feld[DIM];
for(i=0; i < DIM; i++)
 ...
```

Eine Änderung der Dimension des Feldes erfolgt dann an einer einzigen Stelle.

Als „Guter Stil“ hat sich eingebürgert, dass solche Ersetzungen in `GROSSBUCHSTABEN` geschrieben werden, sonstige Variablen dagegen in `kleinbuchstaben`.

Define-Anweisungen können –und sollten– auch in eine Headerfile geschrieben werden.

Beachten Sie auch, dass die Präprozessor-Anweisungen, im Gegensatz zu Programmanweisungen, keinen Strichpunkt enthalten.

Im Präprozessor können auch Bedingungen und Makros definiert werden, darauf soll hier nicht weiter eingegangen werden.

11 Dynamische Datenstrukturen

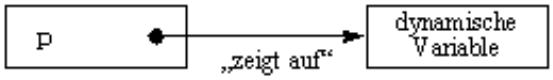
11.1 Dynamische Datenstrukturen

11.1.1 Einführung

Der Aufbau von komplexen Datentypen (Arrays, `struct`, `enum`) hat immer die Eigenschaft, dass er statisch ist. Das bedeutet, dass z.B. Felder immer dieselbe Struktur haben, also aus einer festen Anzahl von Elementen bestehen. Für viele Anwendungen ist dies jedoch eine unzumutbare Einschränkung. Es können nämlich Daten erforderlich sein, die nicht nur ihren Wert, sondern auch ihre Komponenten, ihren Umfang und ihre Struktur ändern können.

Typische Beispiele für solche dynamische Datenstrukturen sind Listen und Bäume. Eine geordnete Liste, z.B. eine alphabetisch geordnete Adressenkartei, könnte als Array realisiert werden. Dies hat jedoch den Nachteil, dass die Länge der Liste vorher festgelegt werden muss, was meist eine Vergeudung von Speicherplatz zur Folge hat. Auch neue Einträge sind dann schwierig, da immer alle folgenden Einträge verschoben werden müssen (aufgrund der Sortierung). Ist die Länge der Liste zu klein gewählt, muss das Programm geändert werden.

Um solche Probleme effizienter lösen zu können, benutzt man Zeiger. Eine Variable `p` vom Typ Zeiger hat als Wert einen Zeiger auf eine dynamische Variable:



Dynamische Variablen werden nicht im Programm deklariert, sondern erst während des Programmablaufs erzeugt. Da sie keine Namen tragen, kann man sie auch nicht über Namen ansprechen, sondern muss über Zeigervariablen auf sie zugreifen.

Der Variablen, auf die der Pointer `p` zeigt, soll 23 zugewiesen werden. Im Vereinbarungsteil muss also nur die Zeigervariable vereinbart werden: als Zeigertyp, und zwar „Zeiger auf `int`“:

```
int *p;
```

Die Variable `p` kann nun auf eine dynamische Variable des Typs `int` zeigen. Diese dynamische Variable ist aber mit der Vereinbarung von `p` noch nicht vorhanden. Auch der Wert von `p` ist zunächst undefiniert . Der Zeiger zeigt also ins „Leere“, bzw. auf irgendeine Speicherzelle.

Dies bedeutet, dass eine Zuweisung der Form `*p = 23;` schlimme Folgen haben kann. Denn möglicherweise zeigt `p` anfangs zufällig auf einen wichtigen Bereich im Speicher, der nun mit einem anderen Wert versehen wird. Oder der Speicher ist zufällig derzeit unbenutzt, wird aber später von einem anderen Programmteil geändert. Die Folge kann ein völlig unkontrollierter Programmablauf sein, die Fehlersuche gestaltet sich dabei ungemein schwierig. Man kann ja kaum zurückverfolgen, welcher Programmteil wann einen falschen Zeiger benutzt hat. Wenn man Glück hat, zeigt der Zeiger auf einen „verbotenen“ Speicherbereich und das Programm stürzt sofort ab. Der Absturz kann aber auch erst viel später, an einer eigentlich korrekten Programmstelle erfolgen, eben als Folge des falschen Zeigerzugriffs.

Achten Sie deshalb unbedingt darauf, dass Sie jeder Zeigervariable einen gültigen Speicherbereich zuweisen, bevor Sie über den Zeiger (die Adresse) auf den Speicherplatz zugreifen.

11.1.2 Speicher reservieren (allozieren) mit `malloc`

Es muss zunächst ein Speicherplatz für die Variable reserviert werden. Um Speicher zu allozieren verwendet man die Funktionen aus der C-Bibliothek. In der Gruppe `stdlib` sind Funktionen vorhanden, mit denen dynamisch Speicher angefordert bzw. wieder frei gegeben werden kann. Um diese Funktionen zu benutzen, muss mit

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

diese Headerdatei eingefügt werden. Damit sind die Funktionen deklariert:

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

`malloc` reserviert Speicherplatz der Größe `size` (der Typ `size_t` ist als positiver Integer definiert), sie liefert einen Zeiger auf diesen Speicherplatz. Da die Größe des benötigten Speicherplatzes für einen bestimmten Datentyp unterschiedlich sein kann, wird der benötigte Platz mit

```
sizeof(Typ)
```

oder

```
sizeof(Variable)
```

ermittelt. Der Zeiger ist vom Typ `void*`, ein generischer Zeiger, der durch Zuweisung in den verwendbaren Typ „Zeiger auf ...“ umgewandelt wird.

Der Aufruf

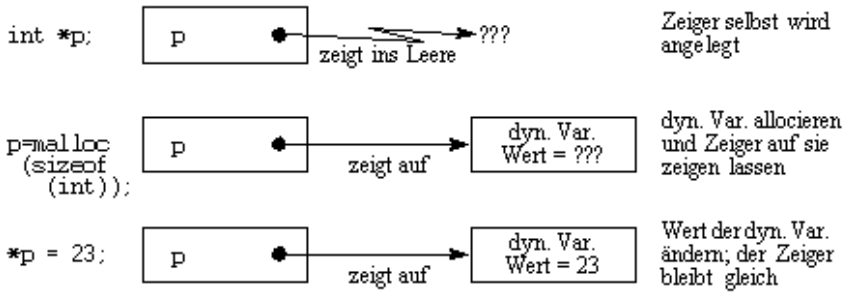
```
int *p;
p = malloc (sizeof(int));
```

bewirkt, dass eine dynamische Variable des Typs `int` erzeugt wird und die Zeigervariable `p` auf diese dynamische Variable zeigt. Die folgende – jetzt erlaubte – Zuweisung belegt die dynamische Variable mit dem Wert 23:

```
*p = 23; /* Weise der dyn. Variablen, auf die p zeigt, den Wert 23 zu */
```

Ist `p` ein Zeiger (Pointer), so ist `*p` der „Inhalt“, auf den `p` zeigt. `*p` ist also die dynamische Variable, auf die `p` zeigt. Und genau diese dynamische Variable wurde gerade mit `malloc(...)` erzeugt.

In den einzelnen Programmzeilen passiert dabei folgendes:



Ein Programm könnte also folgendermaßen aussehen:

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

int main()
{
  int *p1, *p2;
  p1 = malloc (sizeof(int));
                            /*Bezugsvariable generieren*/
  p2 = malloc (sizeof(int));
  if(scanf("%i", p1)==0)    /* int-Wert einlesen */
  {                         /* p1 ist die Adresse der int-Zelle! */
      printf("Unzulaessige Eingabe!");
      return 1;
  }
```

```
  *p2 = *p1 ;                    /* Wert kopieren */
  if(scanf( "%i", p2 )==0)       /* int-Wert einlesen */
  {
      printf("Unzulaessige Eingabe!");
      return 1;
  }
  printf(" %i , %i ", *p1, *p2); /* Werte ausgeben */
}
```

Beim Einlesen mit der Funktion `scanf(...)` kann hier direkt der Zeiger `p1` oder `p2` verwendet werden, da ja bei `scanf` die Adresse der Zelle (also der Pointer auf ...) angegeben werden muss.

Sie müssen nun aber nicht zu jeder Pointer-Variablen einen Speicherplatz mit `malloc` generieren. Sie können auch einer Pointer-Variablen den Wert einer anderen Pointer-Variablen zuweisen:

```
int *p1, *p2;
p1 = malloc (sizeof(int));
p2 = p1;
```

In diesem Fall zeigen `p1` und `p2` auf den selben Speicherplatz für eine `int`-Variable.

Wir ändern das Programm nun etwas ab, indem wir die Zeile `*p2=*p1;` durch `p2=p1;` ersetzen:

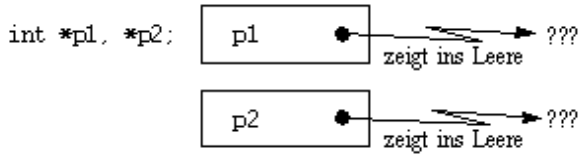
```
#include <stdio.h>
#include <stdlib.h>

int main()
{
  int *p1, *p2;
  p1 = malloc (sizeof(int));
                     /*Bezugsvariable generieren*/
  p2 = malloc (sizeof(int));
  *p1 = 25;          /* int-Wert zuweisen*/
/*  *p2 = *p1 ;        Wert kopieren */
  p2 = p1;           /* p2 zeigt auf die selbe Adresse wie p1 */
  *p2 = 30;          /* weiteren int-Wert zuweisen */
  printf(" %i , %i ", *p1, *p2);  /* Werte ausgeben */
}
```

Hier werden nicht die Werte der dynamischen Variablen kopiert, sondern die Zeiger zugewiesen. Dies hat zur Folge, dass nach der Zeile `p2=p1;` beide Zeiger auf die selbe

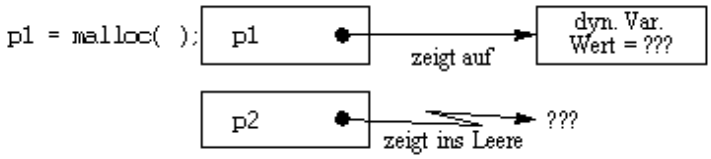
dynamische Variable zeigen. Die dynamische Variable, auf die `p2` vorher gezeigt hatte, ist jetzt *nicht* mehr ansprechbar! Es zeigt kein Pointer mehr auf sie. Der Speicher bleibt aber reserviert und ist somit verloren („memory leak“).

Ausgegeben wird am Ende: `30 , 30`.

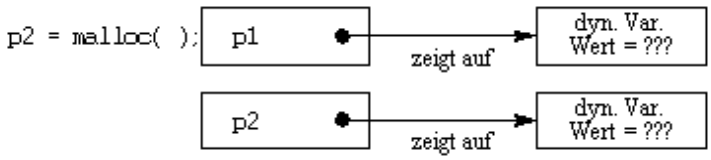
Sehen wir uns den Vorgang nochmal detaillierter an. Zunächst werden die Zeiger selbst vereinbart:



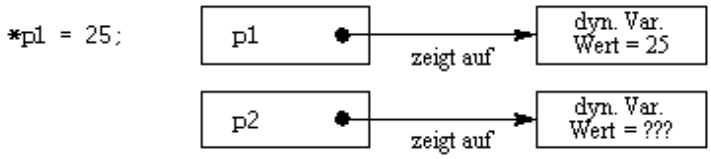
Danach werden die dynamischen Variablen alloziert und die Zeiger so umgebogen, dass sie auf die neu reservierten Speicherzellen zeigen. Als erstes `p1` ...



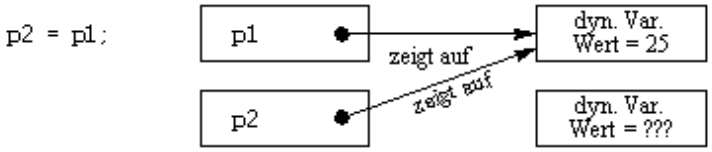
... und dann auch `p2` :



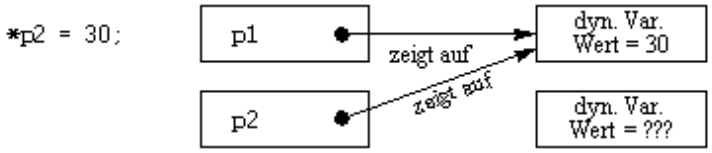
Der dynamischen Variable, auf die `p1` zeigt, wird z.B. der Wert 25 zugewiesen:



Nun wird aber der Zeiger `p2` „verbogen“: ihm wird der Zeiger `p1` zugewiesen, und *nicht* der Wert der dynamischen Variable kopiert. Somit zeigt `p2` nun auf die selbe Adresse, auf die `p1` auch zeigt. Die zweite dynamische Variable (auf die vorher noch `p2` gezeigt hatte) hängt nun „in der Luft“, es gibt keinen Zeiger mehr, der auf sie zeigen würde. Es gibt also keine Möglichkeit mehr, auf ihren Wert zuzugreifen.



Bei der folgenden Wertzuweisung wird nun der Inhalt dieser dynamischen Variable geändert:



Da `p2` aber auf die selbe dynamische Variable zeigt wie `p1`, wird entsprechend auch zweimal der selbe Wert (eben 30) ausgegeben. Hätten wir in beiden Versionen des Programms die Zeile `*p2=30;` weggelassen, so wäre uns der Unterschied evtl. gar nicht aufgefallen: beide Programme würden dann das gleiche Ergebnis ausgeben (`25, 25`), obwohl sie intern doch unterschiedlich sind!

In diesem Fall zeigen `p1` und `p2` auf den selben Speicherplatz für eine `int`-Variable, was in manchen Situationen durchaus sinnvoll und erwünscht sein kann. Andersherum macht das aber weniger Sinn: folgende Anweisungen sind zwar durchaus gültig und erlaubt,

```
int *p;
p = malloc (sizeof(int));
p = malloc (sizeof(int));
p = malloc (sizeof(int));
```

es wird aber dreimal eine neue Bezugsvariable für `p` generiert. Die ersten beiden Bezugsvariablen sind aber nicht mehr erreichbar (es gibt keinen Zeiger auf sie, denn `p` wird jedesmal neu besetzt bzw. „umgebogen“) und liegen somit „tot“ im Speicher herum (das bereits genannte „memory leak“).

11.1.3 Zeiger und Strukturen (`struct`)

Sehen wir uns nun ein neues Beispiel an:

```
enum Monat {Jan,Feb,Mrz,Apr,Mai,Jun,Jul,Aug,Sep,Okt,Nov,Dez};
struct Datum
{
  int tag;
  enum Monat mon;
  unsigned int jahr;
};
```

```
enum Geschlecht {m,w};
struct Person
{
  char vorname[30], nachname[30];
  enum Geschlecht geschlecht;
  struct Datum geburtsdatum;
};

struct Person *p1, *p2;
```

Zeiger des Typs `struct Person*` können nur auf Variablen des Typs `struct Person` zeigen. Die Variablen des Typs `struct Person` sind jedoch, trotz Vereinbarung der Zeiger, noch nicht existent.

Mit welcher Anweisung können wir für die Zeigervariable `p2` eine entsprechende Bezugsvariable erzeugen?

```
p2 = malloc ( sizeof(struct Person) );
```

Auf diese neu erzeugte Variable kann man nun wie gewohnt mit `*p2` zugreifen. Das hilft uns aber meist noch nicht weiter, wir wollen ja auf die einzelnen Komponenten zugreifen. Dieser Zugriff erfolgt wie gewohnt über den Punkt, also um auf den Nachnamen zuzugreifen kann man schreiben:

```
(*p2).nachname
```

Dabei ist die Klammer `(*p2)` wichtig, da die Priorität des Punkts `"."` höher ist als die des Dereferenzierungsoperators `"* "`. Diese Schreibweise ist aber etwas umständlich. Weil derartige Zugriffe in C aber recht häufig vorkommen, wurde extra eine gleichwertige, einfachere Schreibweise eingeführt:

```
p2->nachname
```

Hat man einen Pointer auf eine Struktur, so benutzt man den Operator `->` für den Zugriff auf die Komponenten der Struktur. Dies mag auf den ersten Blick etwas ungewöhnlich erscheinen, ist aber nach kurzer Eingewöhnung wesentlich übersichtlicher.

Wie kann man das Geschlecht der Person `p2` auf weiblich setzen?

```
p2->geschlecht = w;
```

Und wie setzen Sie das Geburtsjahr auf 1981?

```
p2->geburtsdatum.jahr = 1981;
```

Hier wurden also `->` und `.` gemischt verwendet. Das ist auch richtig so, da ja nur `p2` ein Zeiger ist. Die Komponente `geburtsdatum` ist selbst eine Struktur, und eben kein Zeiger, daher wird auf ihre Komponenten wie gewohnt mit dem Punktoperator `.` zugegriffen.

```
#include <stdio.h>                  /* nötig für Ein-/Ausgabe */
#include <stdlib.h>

int main()                          /* Programm-Kopf */
{
  int i, j;
  enum Monat{Jan,Feb,Mrz,Apr,Mai,Jun,Jul,Aug,Sep,Okt,Nov,Dez};/*Aufzählungstyp*/
  enum Art {w,m};
  enum Tag {Mo,Di,Mi,Do,Fr,Sa,So};
  struct Datum /* Struktur-Vereinbarung */
  {
   int tag;
   enum Monat mon;
   unsigned int jahr;
  };

  struct Person /* Struktur-Vereinbarung */
  {
    char vorname[31], nachname[31];
    enum Art geschlecht;
    struct Datum geburtsdatum;
  };

  struct Person *p1, *p2, *pTmp;

  p1 = (struct Person*) malloc ( sizeof(struct Person) );   /* Speicher für p1
allozieren */
  p2 = (struct Person*) malloc ( sizeof(struct Person) );   /* Speicher für p2
allozieren */

  for(i=0; i < 2; i++)
  {
    if (!i)   /*falls  i == 0*/
      pTmp = p1;
    else
      pTmp = p2;
    printf ("Vorname %d? ", i+1);
    if  (scanf("%s", pTmp->vorname)==0)
    {
        printf ("Unzulaessige Eingabe!");
```

```
        return 1;
    }
    /* funktioniert nur bis zu einer Länge von 30->Arraygröße! */

    printf ("Nachname %d? ", i+1);
    if (scanf("%s", pTmp->nachname)==0)
    {
        printf ("Unzulaessige Eingabe!");
        return 1;
    }
    /* funktioniert nur bis zu einer Länge von 30->Arraygröße! */

    printf ("Geburtstag %d? ", i+1);
    if (scanf(" %d", &pTmp->geburtsdatum.tag)==0)
    {
        printf ("Unzulaessige Eingabe!");
        return 1;
    }

    printf ("Geburtsmonat %d? ", i+1);
    if (scanf(" %d", &j)==0)
    {
        printf ("Unzulaessige Eingabe!");
        return 1;
    }
    /* Benutzer sollte 1-12 eingeben */

    pTmp->geburtsdatum.mon = (enum Monat)(j-1);
    /* enum Monat kennt Werte von 0-11 */

    printf ("Geburtsjahr %d? ", i+1);
    if (scanf(" %u", &pTmp->geburtsdatum.jahr)==0)
    {
        printf ("Unzulaessige Eingabe!");
        return 1;
    }

    printf ("Geschlecht %d [weiblich: 1, maenlich: 2]? ", i +1);
    if (scanf(" %d", &j)==0)
    {
        printf ("Unzulaessige Eingabe!");
        return 1;
    }
    /* Benutzer sollte 1(weiblich) oder 2(männlich) eingeben */
    pTmp->geschlecht = (enum Art)(j-1); /*enum Art kennt Art von 0-1*/
}/*for*/
```

```
    printf ("Folgende Daten wurden eingegeben:\n");
    for(i=0; i <2; i++)
    {
      if (!i)  /* falls  i == 0 */
        pTmp = p1;
      else
        pTmp = p2;
      if (pTmp->geschlecht == w)
        printf ("Frau\n");
      else
        printf ("Herr\n");
      if (pTmp->geburtsdatum.mon < Jul)
        printf ("%s %s geboren in der ersten Jahreshaelfte (%d.%d.) %d.\n",
                pTmp->vorname, pTmp->nachname, pTmp->geburtsdatum.tag,
                pTmp->geburtsdatum.mon+1, pTmp->geburtsdatum.jahr);
      else
        printf ("%s %s geboren in der zweiten Jahreshaelfte (%d.%d.) %d.\n",
                pTmp->vorname, pTmp->nachname, pTmp->geburtsdatum.tag,
                pTmp->geburtsdatum.mon+1, pTmp->geburtsdatum.jahr);
      }/* for */
      return 0;
  }/* main */
```

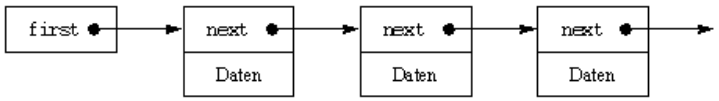
Listing 11.1: struct mit malloc
Das Programm hat die gleiche Funktionalität wie *struct*, das bereits in Kapitel 8.4 vorgestellt wurde. *struct mit malloc* arbeitet aber mit Zeigern und alloziert den für die Strukturen benötigten Speicher mittels `malloc` zur Laufzeit.

11.1.4 Verkettete Listen

Zurück zu unserem Beispiel:

```
...
struct Person
{
  char vorname[30], nachname[30];
  enum Geschlecht geschlecht;
  struct Datum geburtsdatum;
};

Person *p1, *p2;
```

Die Tatsache, dass Variablen vom Typ Zeiger nicht nur auf Standardvariablen zeigen können, sondern dass dies auch Strukturen sein können, die selbst wieder Zeiger enthalten, macht die Verwendung von Zeigern besonders nützlich. Ein Beispiel für eine verkettete Liste:



Eine verkettete Liste besteht also aus einem Anker (`first`) und eventuell mehreren Elementen vom Typ einer Struktur. Um die Elemente miteinander verketten zu können, enthält jedes Listenelement außer den eigentlichen Daten eine Komponente vom Typ Zeiger, die als Verweis auf das jeweils nächste Element in der Liste dient. Jedes einzelne der Listenelemente ist dabei eine dynamische Variable, die zur Laufzeit (mit `malloc`) alloziert wird.

Die Typdefinition der Struktur sieht demnach so aus:

```
struct Element
{
  struct Element *next;
  <irgendein Typ> Daten; /* beliebig weitere */
};
```

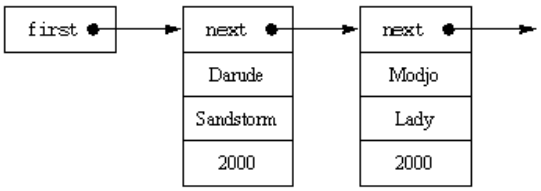
Eine verkettete Liste ist während des Programmlaufs (also dynamisch) erweiterbar. Sie wird deshalb als lineare Datenstruktur (Folge von Elementen) eingesetzt, deren Länge beim Entwurf des Programms noch nicht feststeht, sondern während des Programmlaufs variieren kann. Man kann z.B. in einer Schleife immer mehr Elemente allozieren und in die Liste „einhängen“, obwohl die Anzahl der Elemente erst während der Ausführung des Programms klar wird.

Verkettete Listen belegen außerdem immer nur den ihrer aktuellen Länge entsprechenden Speicherplatz, während Arrays den in der Deklaration statisch vereinbarten Speicherplatz benötigen. Ebenso lassen sich Änderungen (z.B. Einfügen eines Elements) mit geringerem Aufwand durchführen, da nur Zeiger „umzuhängen“ sind, während in Arrays ganze Bereiche umkopiert werden müssen.

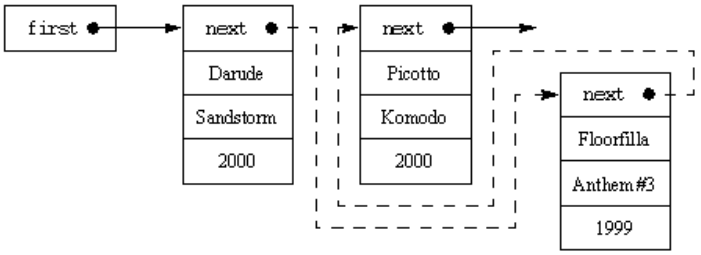
Sehen wir uns einfach mal ein Beispiel an. Sie wollen eine sortierte Liste der Musikstücke auf Ihrem Computer erstellen. Von jedem Stück sollen Interpret und Titel, sowie Erscheinungsjahr gespeichert werden. Für Interpret und Titel sollen 30 Zeichen genügen. Da die Anzahl der Musikstücke laufend wächst, sollen sie in einer verketteten Liste abgelegt werden. Sie definieren also folgende Struktur:

```
struct Stueck
{
  struct Stueck *next;  /* Zeiger auf naechstes Stueck */
  char interpret[31];   /* 30 Zeichen plus Null-Byte */
  char titel[31];       /* ebenfalls 30 Zeichen */
  int jahr;             /* Erscheinungsjahr */
};
```

Jedes `struct Stueck` wird dann als Element einer verketteten Liste im Speicher abgelegt; diese Liste kann z.B. nach Interpreten sortiert sein:



Sie wollen nun ein neues Stück in diese Liste aufnehmen. Sie erzeugen den Speicherplatz (mit `malloc`) und wollen das Element nun an der richtigen Stelle in der Liste einhängen:



Wie die gestrichelten Linien zeigen, genügt es, zwei Zeiger passend zu setzen, und schon ist das neue Stück in der Liste enthalten. Der tatsächliche Speicherort ist dabei unwichtig, da die Liste ja nur entsprechend ihrer `next`-Zeiger durchgegangen wird.

Zwei wichtige Fragen zu verketteten Listen wurde bislang noch nicht gestellt.

Wo ist der Anfang der Liste? Bei verketteten Listen ist es wichtig, dass man den Zeiger auf das erste Element der Liste in einer Variable abspeichert:

```
struct Stueck *first;
```

Verliert man diesen Zeiger, so besteht kein Zugang mehr zu der gesamten Liste! Um die Elemente einer Liste durchzugehen, sollte man also einen anderen Zeiger verwenden. Wir haben ja schon gesagt, dass mehrere Zeiger auf die selbe Bezugsvariable zeigen dürfen. Hier ist also einer der Anwendungsfälle.

Die zweite offene Frage wäre:

Wie erkennt man das Ende der Liste? Da die Liste irgendwann zu Ende ist, darf der Zeiger im letzten Element auf kein weiteres Element mehr verweisen. Hierfür gibt es einen speziellen Pointer: den `NULL`-Zeiger. Dieser Zeiger zeigt in C immer auf eine ungültige Speicheradresse. Nimmt man für die Adresse die Zahl 0, so wird dieser immer als

NULL-Zeiger gewertet. Man kann sich vorstellen, dass der NULL-Zeiger auf die Adresse Null zeigt[1].

Somit kann man für das letzte Element einfach next=0 schreiben. Viele Headerfiles (*stdio.h* und *stdlib.h*) definieren die Konstante NULL als die Zahl 0, so dass man besser schreiben kann:

```
next = NULL;
```

Hat eine Pointer-Variable den Wert NULL, dann besitzt sie keine Bezugsvariable. Wenn also die Pointer-Variable p den Wert NULL besitzt, ist die Formulierung *p unsinnig, da p auf keinen sinnvollen Speicherplatz verweist.

Die Wahl der Adresse NULL hat dabei einen großen Vorteil: 0 gilt in logischen Ausdrücken als nein (FALSE), während jeder andere Wert als ja (TRUE) gewertet wird. Dies heißt, dass man den Zeiger auf das nächste Element z.B. in einer if-Abfrage direkt als logischen Ausdruck verwenden kann; wenn der Zeiger auf 0 zeigt, ist der Ausdruck FALSE, zeigt er auf irgend eine andere Adresse, ist er TRUE. Dies kann man z.B. ausnutzen, um die gesamte Liste durchzugehen und alle Einträge auszugeben. Sei first der Zeiger auf das erste Element (Anker), und p ein weiterer Zeiger vom Typ struct Stueck *, so kann man folgendes schreiben:

```
p = first;

while (p)
{
  printf("%s : %s", p->interpret, p->titel);
  printf("(%d)\n", p->jahr );
  p=p->next;
}
```

Zunächst wird hier der Zeiger p (als Hilfszeiger) auf das erste Element der verketteten Liste gesetzt. In der while-Schleife wird dann geprüft, ob dieser Zeiger überhaupt noch auf eine gültige Adresse zeigt. Solange dies der Fall ist, wird das aktuelle Stück ausgegeben und p auf das nächste Element weitergeschaltet.

Diese Schleife könnte man sogar mit einer for-Schleife realisieren; p wird hier zum Schleifenzähler:

```
for (p=first; p; p=p->next)
{
  printf("%s : %s", p->interpret, p->titel);
```

[1] Ein NULL-Zeiger muss nicht immer wirklich auf die Adresse 0 zeigen, es gibt tatsächlich einige Computersysteme, in denen eine andere Adressen als NULL-Zeiger verwendet wird. Der Programmierer muss sich mit diesen Feinheiten aber meist nicht auseinandersetzen, da der Compiler NULL beim Übersetzen in die passende Adresse umwandelt.

```
  printf("(%d)\n", p->jahr );
}
```

11.1.5 Speicher mit `free` wieder freigeben

Wir haben jetzt gesehen, wie man dynamisch immer mehr Speicher allozieren kann und diesen z.B. in einer verketteten Liste verwaltet. Immer nur mehr Speicher reservieren führt natürlich dazu, dass unser Programm irgendwann an die Grenzen des verfügbaren Speichers stößt. Es muss also auch noch eine Möglichkeit geben, bereits allozierten Speicher wieder freizugeben.

Genau dies erledigt die Funktion `free()`. Der Aufruf der Funktion `free(p);` bewirkt, dass der Speicherbereich der Variable, auf die `p` zeigt, wieder freigegeben wird.

Die folgende Anweisungsfolge löscht das erste Element der Liste:

```
p=first;
first=p->next;
free (p);
```

Zunächst wird der Hilfszeiger `p` auf das erste Element der Liste gesetzt (um deren Adresse zwischenzuspeichern). Durch die Zuweisung `first=p->next;` rückt das zweite Element der Liste an die erste Stelle auf (`first` zeigt auf den Anfang der Liste). Mit `free(p)` wird das ursprünglich erste Element gelöscht (freigegeben).

Bei der Verwendung von `free(p);` ist jedoch Vorsicht geboten! Wurde nämlich der Zeiger `p` vor `free(p);` einer anderen Variablen `p1` zugewiesen, zeigt nun auch `p1` auf einen nicht mehr existierenden Speicherplatz. Zudem darf `free` nur auf Objekte angewandt werden, die mit `malloc` erzeugt wurden.

Zu beachten ist ferner: `malloc` und `free` sind Funktionen aus der Standardbibliothek, die wie jede Funktion vor der Verwendung deklariert werden müssen. Dies erfolgt durch:

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

Ein kleines Beispiel zu Pointern:

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

int main()
{
  char *meinname;                        /* meinname zeigt auf einen String */
  int *meinalter;                        /* Zeiger auf eine int-Variable */

  meinname = malloc(80 * sizeof(char));  /* Platz für 80 Zeichen */
```

```
  meinalter = malloc (sizeof(int));
  *meinalter = 24;
  strcpy (meinname, "Max Mustermann");

  printf ("Mein Name ist %s ", meinname);
  printf ("und ich bin %2d Jahre alt.", *meinalter);

  free(meinalter);
  free(meinname);
}
```

Ein weiteres Beispiel (Achtung, ist nicht wirklich sinnvoll!):

```
/* Beispiel f. die Kombination von Pointer und Strukturen */

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

int main()
{
  struct datum
  {
    unsigned int tag, monat, jahr;
  };
  struct vollstname
  {
    char vorname[41], nachname[41];
  };
  struct person
  {
    struct vollstname  name;
    char ort[21], staat[21];
    struct datum geburtstag;
  };

  struct person *selbst, *ident, *extra;
  selbst = (struct person*) malloc(sizeof(struct person));

  strcpy(selbst->name.vorname, "Wolfgang");
  strcpy(selbst->name.nachname, "Maier");
  strcpy(selbst->ort, "Irgendwo");
```

```
    strcpy(selbst->staat, "Deutschland");
    selbst->geburtstag.tag = 28;
    selbst->geburtstag.monat = 10;
    selbst->geburtstag.jahr = 1986;
    ident = selbst;
    /* ident zeigt auf die gleiche Speicheradresse wie selbst */

    extra = (struct person*) malloc(sizeof(struct person));
    *extra = *selbst;
    /* die Struktur, auf die selbst zeigt, wird der
       Struktur zugewiesen, auf die extra zeigt
    */

    printf("%s ",selbst->name.vorname);
    printf("%s \n",selbst->name.nachname);
    printf("%s ",ident->ort);
    printf("%s \n",ident->staat);
    printf("%d. ",extra->geburtstag.tag);
    printf("%d. ",extra->geburtstag.monat);
    printf("%d \n",extra->geburtstag.jahr);

    free(selbst);
    /* ident zeigt auf den identischen Speicherbereich
       -> kein weiteres free noetig
    */
    free(extra);
}
```

11.1.6 Häufige Fehler

- Zuweisen von Zeigern anstelle der Werte (`iptr=jptr` statt `*iptr=*jptr`), wenn Werte kopiert werden sollen. Das Problem tritt häufig erst später in Erscheinung, da zunächst die Zuweisung erfolgreich scheint: nach `iptr=jptr` gilt ja auch `*iptr==*jptr`)
- Zugriff auf nicht allozierten Speicher. Kann zu sofortigem Absturz führen, meist treten die Probleme aber erst später auf und lassen sich dann nur schwer zuordnen.
- Zeiger auf eine automatische lokale Variable als Rückgabewert zurückgeben (oder an eine globale Variable zuweisen). Nach dem Ende der Funktion existiert die automatische lokale Variable nicht mehr und der Zeiger zeigt ins Leere.

- Freigeben eines vorher nicht allozierten Speicherbereichs. Kann sich evtl. auch erst viel später auswirken.
- Mehrfaches Freigeben (auch wenn zwei Zeiger auf den selben Speicherbereich zeigen, darf der Speicher nur einmal freigegeben werden) - ebenso unvorhersehbare Folgen.
- Allozierten, aber nicht mehr benötigten Speicher nicht freigeben („memory leak“). Meist kein größeres Problem, da der Speicher am Programmende automatisch freigegeben wird, kann aber durch Verschwendung des gesamten allozierbaren Speichers zu Programmabsturz führen.

11.2 Vertiefung: Zeiger, ein mächtiges Werkzeug

Das Programmieren mit Zeigern ist wohl einer der schwierigeren Aspekte von C. Wir wollen daher noch etwas tiefer in die „Innereien“ der Zeiger blicken, in der Hoffnung, damit auch ein tieferes Verständnis für deren Welt zu erzeugen. Wir beginnen hierfür mit den „einfachen“ Variablen.

11.2.1 Variablen

Wir haben ja bereits im Kapitel 10.5.2 gesehen, dass ein Compiler eine normale Variable in einer Speicherzelle ablegt. Der Inhalt dieser Speicherzelle ist der Wert unserer Variable, die Adresse der Speicherzelle entspricht dem Namen der Variable. Wir haben den Namen als Pseudonym für die Speicheradresse bezeichnet. In Abbildung 11.1 sehen Sie nochmal diesen Zusammenhang für die Variable `mych`, von der wir einfach annehmen, dass sie an der Adresse 500 gespeichert wird.

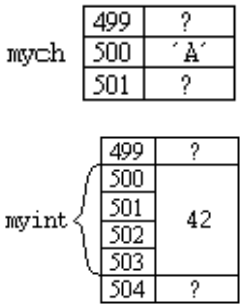


Abbildung 11.1: Das Speicher-Layout von bekannten Datentypen.

```
char mych = 'A';
```

Nicht jede Variable belegt aber nur eine einzelne Speicherzelle. Wir hatten ja verschiedene Variablentypen kennengelernt. Auf vielen Rechner-Architekturen belegt z.B. ein int insgesamt vier (aufeinanderfolgende) Speicherzellen, um einen größeren Wertebereich darstellen zu können:

```
int myint = 42;
```

Bei Strukturen können es durchaus auch z.B. mehrere Hundert Speicherzellen sein, die von einer (Struktur-) Variable belegt werden. Die Variable selbst ist dabei immer noch ein Pseudonym für die Speicheradresse (hier einfach die Adresse der ersten Zelle, die

zur Variable gehört). Der Typ der Variablen gibt schließlich Auskunft über die Anzahl an Speicherzellen, die zu ihr gehören, und wie diese Zellen zusammenhängen.

Eine einzelne Variable hat auch immer ihre feste Größe. Wenn man nun größere Speicherbereiche nutzen will, kann man Datenfelder (Arrays) einsetzen.

11.2.2 Felder

Arrays (Felder, Vektoren) haben zwar wie einfache Variablen auch eine feste Größe, aber man kann sich ihre Größe immerhin (beim Schreiben des Programms) selbst aussuchen. Bei eindimensionalen Arrays liegen nun die einzelnen Elemente einfach hintereinander im Speicher, wie in Abbildung 11.2 gezeigt:

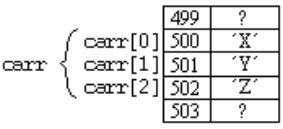


Abbildung 11.2: Elemente eines Feldes bilden eine Sequenz im Speicher.

```
char carr[3];
carr[0]='X';
carr[1]='Y';
carr[2]='Z';
```

Interessanterweise speichert C nirgends die Größe des Arrays. Der Typ des Arrays gibt nur Auskunft über die Größen der einzelnen Elemente (hier jeweils eine Zelle: es ist ein `char`-Array). Die Array-Variable ist wieder ein Pseudonym für die erste betroffene Speicherzelle, der Rest des Arrays liegt dann dahinter. Die Größenangabe im Programm (bei der Array-Vereinbarung) wird nur vom Compiler benutzt, um ausreichen Platz freizuhalten. Es ist die Aufgabe des Programmierers, auch wirklich nur den vereinbarten Bereich zu nutzen.

Die Tatsache, dass nirgends die Länge vermerkt wird, kann zu sehr obskuren Fehlern führen: wenn man auf Elemente zugreifen will, die es im Array in Wirklichkeit gar nicht mehr gibt („out of bounds“), liest oder ändert man irgendeine unbekannte Speicherzelle. Je nachdem, welche Aufgabe diese Zelle eigentlich hat, wird sich das Programm früher oder später anders verhalten als erwartet; wenn man Glück hat, stürzt es sofort ab. Da die Anordnung der Variablen im Speicher eine Aufgabe des Compilers ist, wird sich so ein fehlerhaftes Programm auch je nach Compiler und Rechnerarchitektur unterschiedlich verhalten.

Die Abbildung des Array-Index auf die Speicheradresse ist bei „längeren“ Elementvariablen nicht ganz so einfach: hier ist noch die Größe der einzelnen Elemente zu Berücksichtigen. Die einzelnen Elemente liegen immer noch hintereinander im Speicher, aber die Adressen haben einen größeren Abstand (um genügend Platz für den Inhalt zu haben). Vergleichen Sie den folgenden Code mit Abbildung 11.3 auf der nächsten Seite.

```
int iarr[2];
iarr[0]=47;
iarr[1]=11;
```

Die Größe der Elemente (und deren Typ) ist aus dem Typ des Arrays ersichtlich.

11.2.3 Zeiger

Aber was hilft uns dieses Wissen nun bei der Betrachtung von Zeigern? Nun, Pointer sind ja Zeiger auf Speicherbereiche, sie enthalten also eine Adresse. Ein Pointer muss aber selbst auch wieder abgespeichert werden – sonst könnte man mit der Adresse ja nichts anfangen. Nehmen wir also den folgenden Zeiger unter die Lupe:

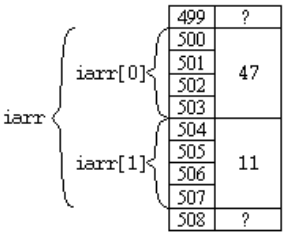


Abbildung 11.3: Das Speicher-Layout von `iarr`.

```
char *chptr;
char mych='M';
chptr = &mych;
```

Wie sieht das nun im Speicher aus? Abbildung 11.4 zeigt, dass `chptr` und `mych` natürlich beide im Speicher abgelegt werden, jeder an einer eigenen Adresse (hier beispielhaft 500 und 600). Während in `mych` aber wie gewohnt direkt der Wert der Variablen steht (`'M'`), enthält `chptr` eine Speicheradresse. Wir haben mit `chptr = &mych;` dem Zeiger die Adresse der Variablen `mych` zugeweisen, also enthält er auch deren Adresse, im Beispiel eben 600.

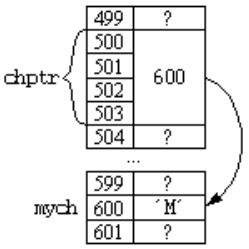


Abbildung 11.4: Zeiger sind Variablen, in die man Speicheradressen speichern kann.

Nun sollte auch der Unterschied zwischen direkter Benutzung von `chptr` und Verwendung von `*chptr` (Dereferenzierung) deutlicher werden: während mit `chptr` der Zeiger selbst gemeint ist (also Speicherzellen 500 bis 503), so greift man mit `*chptr` auf die Speicherzelle zu, deren Adresse im Zeiger steht, hier auf die 600. An Adresse 600 ist hier die Speicherzelle, auf die unser Pointer zeigt.

Ein Zeiger kann natürlich auch für beliebige andere Typen von Variablen vereinbart werden, nehmen wir mal eine Integer-Variable (Abbildung 11.5 auf der nächsten Seite):

```
int *iptr;
int myi=2065;
iptr = &myi;
```

Der Unterschied liegt unter anderem in der Anzahl an Speicherzellen, die ein `int` belegt. Der Zeiger selbst (hier `iptr`) bleibt genauso groß wie im vorausgehenden Beispiel, er muss ja immer noch eine Adresse speichern, nicht mehr und nicht weniger. Die Variable `myi` benötigt aber mehr Speicher, und der Inhalt der einzelnen Speicherzellen bedeutet natürlich auch etwas Anderes als wenn es sich um `char`-Variablen handeln würde.

Solange man über die Variable `myi` auf die Speicherzelle(n) zugreift, ist das ja auch kein Problem, der Typ der Variable sagt aus, wie der Speicherinhalt zu deuten ist und wieviele Zellen belegt sind. Beim Zugriff über `*iptr` muss das aber auch klar sein: aus genau diesem Grund sind Zeiger nicht einfach nur vom Typ Zeiger, sondern vom Typ `char`-Zeiger (`char *`) oder wie hier `int`-Zeiger (`int *`).

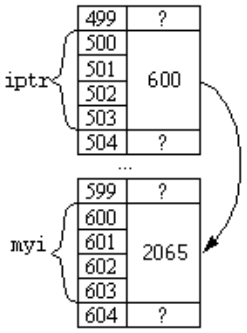


Abbildung 11.5: Ein Zeiger auf einen int-Wert.

11.2.4 Dynamische Speicherallozierung

So richtig interessant werden Zeiger im Zusammenhang mit dynamischer Speicherallozierung. Hier weist man dem Zeiger nicht mehr die Adresse einer vorher sowieso schon bekannten Variable zu, sondern fordert zusätzlichen Speicher an und lässt den Pointer auf diesen zusätzlichen Speicher zeigen.

Dieses Anfordern von zusätzlichem Speicher wird auch Reservieren oder Allozieren von Speicher genannt. Natürlich wird dabei nicht der physikalische Speicher vergrößert, aber vereinfacht gesagt wird dem Programm vom Betriebssystem weiterer Speicher zugeteilt, der vorher unbenutzt war.

Es wird während der Ausführung des (übersetzten) Programms entschieden, dass (und wie viele) weitere Speicherzellen benötigt werden. Mit dem Aufruf von `malloc` fordert man Speicher an und erhält einen Zeiger auf diesen zusätzlichen Speicher.

```
char *nptr;
nptr = malloc(sizeof(char));
*nptr='F'
```

Im Beispiel von Abbildung 11.6 befindet sich der neu allozierte Speicher an Adresse 900. Diese Adresse wird von `malloc(sizeof(char));` zurückgegeben und dem Zeiger `nptr` zugewiesen. Über diesen Zeiger wird die Speicherzelle schließlich mit dem Wert `'F'` initialisiert.

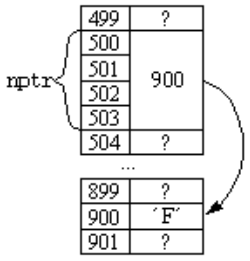


Abbildung 11.6: Zeiger und dynamische Speicherallozierung

Wie man im Bild auch erkennen kann, ist der Zeiger die einzige Möglichkeit, auf den neuen Speicher auch zuzugreifen. Diese neu Speicherzelle hat keinen anderen Namen, sie wird also nicht über eine normale Variable angesprochen. Der Speicher bleibt so lange reserviert, bis das Programm endet, oder der Programmierer den Bereich mit `free()` wieder freigibt. Nach der Freigabe mit `free()` kann diese Speicherzelle wieder an anderer Stelle verwendet werden.

Natürlich muss das Betriebssystem den aktuellen „Belegungszustand“ der Speicherzellen verwalten: wenn das Programm neuen Speicher anfordert, muss ja irgendwo be-

kannt sein, welche Bereiche reserviert sind und welche noch frei sind. Diese Information steht natürlich auch im Speicher, häufig unmittelbar vor jedem reservierten Bereich. Wenn man nun bei der Freigabe eine falsche Adresse angibt, oder einen schon freigegebenen Bereich ein zweites mal freigeben will, dann stimmen diese Verwaltungsdaten nicht mehr und das Programm wird sich unerwartet verhalten, oder auch wieder abstürzen. Ähnlich ist es bei Zugriffen auf Pointer, die in unbekannte oder schon wieder freigegebene Speicherbereiche zeigen.

Schauen Sie sich nun zum Thema Zeiger noch einmal das Kapitel 11.1.6 an. Es sollte nun klar sein, warum jeder dieser Punkte so gravierende Folgen haben kann.

11.2.5 Funktionen und Zeiger

Es wurde ja bereits besprochen, wie der Datenaustausch zwischen Funktionen und dem aufrufenden Programm in C bewerkstelligt wird. Über die Parameterliste kann eine beliebige Anzahl von Daten mit vereinbartem Typ als Eingabe in die Funktion übergeben werden. Der Returnwert ist ein einzelner Wert vom Typ der Funktion.

Für Ein- und Ausgabe können alle bekannten Datentypen verwendet werden. Der Typ `void` steht in der Parameterliste bei einer Funktion ohne Eingabeparameter, bzw. als Return-Typ bei einer Funktion ohne Rückgabewert.

Zusätzlich zu den Standarddatentypen von C können für Ein- und Ausgabe auch selbst definierte Datentypen verwendet werden. In beiden Fällen (als Parameter oder Returntyp) können auch Zeiger verwendet werden, man verwendet also nicht den Wert, sondern den Zeiger auf den Wert.

Bei komplexen Datentypen, wie Feldern und Strukturen, wird nicht das gesamte Feld übergeben, sondern man übergibt hier nur einen Wert, die Adresse der Daten, also den Zeiger. Damit können flexible Funktionen geschrieben werden, die auf umfangreiche Daten im Programm zugreifen, sie lesen oder auch verändern können.

Auch für einfache Datentypen kann über die Adresse, den Zeiger gearbeitet werden. Damit kann die Funktion auf die Daten im aufrufenden Programm zugreifen und sie modifizieren. Als Parameter wird ein „Zeiger auf ...“ übergeben.

```
void funkt (int *iptr)
{
  ...
  *iptr = ...
  ...
  return;
}

int wert;
...
/* Aufruf von funkt, Übergabe des Zeigers auf Wert */
```

```
funkt (&wert);
/* die Funktion funkt kann den Inhalt der Zelle wert ändern */
...
```

Damit hat man also einen „Referenzparameter“, den es in ANSI-C eigentlich nicht gibt; es wird hier ja auch nur eine Adresse „by-value“ übergeben. Damit hat die Funktion Zugriff, sowohl lesend als auch schreibend, auf die angegebenen Speicherstellen. Dies entspricht dem Vorgehen, wie es auch bei der Übergabe von Feldern an Funktionen verwendet wird.

Wir wollen noch mal auf die Eingabe-Funktion `scanf` zurückkommen. Warum gibt es in C keine einfache Lesefunktion `read(a)` zum Einlesen eines Wertes in die Zelle `a`, wie dies in anderen Sprachen möglich ist? Der Grund ist, dass es in C nur einen value-Parameter gibt, mit dem man einen Wert (by value) an die Funktion übergeben kann. Der reference-Parameter, mit dem man aus einer Funktion über die Parameterliste (by reference) zurückgeben kann, fehlt. Hier muss mit Zeigern gearbeitet werden. Ein Zeiger wird an die Funktion übergeben, über diesen Zeiger kann die Funktion auf die dazugehörige Zelle zugreifen und einen Wert ablegen. Aus diesem Grund erwartet die Funktion `scanf` einen Zeiger, eine Adresse, um dort den eingelesenen Wert abzulegen.

```
int a, *pa; scanf ( " %d" , &a ) ;
```

mit

```
pa = &a;
```

kann hier natürlich auch die Zeigervariable verwendet werden:

```
scanf ("%d", pa );
```

Eine weitere Möglichkeit ist es, einer Funktion die Berechnungsvorschrift erst beim Aufruf zu übergeben, also beim Aufruf anzugeben welche Funktion verwendet werden soll. Auch hier wird mit Zeigern gearbeitet. Der Name der Funktion ist der Pointer auf die Funktion.

Sehen Sie sich das folgende Beispiel an:

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

int main()
{
  double x,y,z;
  double trig (double (*wie)(double));
                    /* wie ist die Berechnungsvorschrift, der Funktionsname */
```

```
  double tst (double wi);
  x = sin(1.);
  y = cos(1.);
  z = tan(1.);
  printf ("sin(1): %f cos(1): %f tan(1): %f \n", x, y, z);
  x = trig(sin);    /* die Funktion trig wird mit sin()*/
  y = trig(cos);    /* bzw. cos()*/
  z = trig(tst);    /* oder der eigenen Funktion tst() aufgerufen */
  printf ("sin(1): %f cos(1): %f tan(1): %f \n",x,y,z);
}

/* Die Funktion trig hat einen Eingabeparameter:
   eine double-Funktion mit einem double-Parameter */

double trig (double (*wie)(double))
{
  double a;
  a = (*wie)(1.);
  return a;
}

double tst (double wi) /* tst == tan == sin/cos */
{
  return (sin(wi)/cos(wi));
}
```

11.2.6 Zeiger und Felder, Adressrechnung

Felder bestehen aus mehreren Elementen des gleichen Datentyps. Zunächst wollen wir uns auf eindimensionale Felder, also Vektoren beschränken. In einem Feld haben, im Gegensatz zu einer Struktur, alle Elemente den gleichen Datentyp. Datentyp und Größe des Feldes werden bei der Deklaration vereinbart.

z.B.

```
int feld [100]; /* Ein Feld mit 100 Elementen vom Typ Integer */
```

Hierbei können neben den vorhandenen Grunddatentypen auch die selbst vereinbarten Datentypen wie Strukturen und Aufzählungstyp verwendet werden. Die Größe des Feldes, also die Anzahl der Elemente muss bei der Deklaration fest vereinbart werden, hier sind keine variablen Angaben möglich. Der Zugriff auf die einzelnen Elemente erfolgt über den Index. Der Index läuft von 0 bis N-1, also in unserem Beispiel von 0 bis 99 , damit ist das erste Element `feld[0]`, das letzte Element `feld[99]`. Hier kann der

Index natürlich auch eine Variable oder ein ganzzahliger Ausdruck sein. Auf jedes Element kann man, wie auch bei Variablen, über die Adresse (den Zeiger) zugreifen. Mit der Vereinbarung `int * iptr;` und `iptr=&feld[0];` greift man mit `*iptr = 123;` ebenso auf das erste Element zu; oder auf ein beliebiges Element `feld[i]` kann man mit `iptr=&feld[i];` und `*iptr = 123;` zugreifen. Die einzelnen Elemente eines Feldes werden im Speicher hintereinander abgelegt. Die Adresse des Feldes ist damit die Adresse des ersten Elementes.

Wie erhält man die Adresse des ersten Elementes? Die Adresse des ersten Elementes ist `iptr = &feld[0];` oder einfacher `iptr = feld;`. Die Adresse des Feldes (des 1. Elementes) ist also der Feldname, ohne Index.

Mit dieser Adresse, mit diesem Zeiger, kann man nun auf jedes Element des Feldes zugreifen, indem der Offset vom Anfang zu der Adresse addiert wird, also erfolgt mit `*(iptr+25)` der Zugriff auf `feld[25]` bzw. allgemein greift `*(iptr+i)` auf `feld[i]` zu. Ebenso kann auch ohne eine Zeigervariable direkt `*feld` oder `*(feld+i)` verwendet werden. Da nach der Übersetzung des Programms nur noch mit Adressen gearbeitet wird, ist der Zugriff über den Index oder über den Zeiger (+ Offset) völlig identisch. Will man das gesamte Feld bearbeiten so kann man sowohl den Index- als auch den Zeigerzugriff in eine Schleife packen. In jedem Fall muss man selbst darauf achten, dass die Feldgrenzen nicht über- oder unterschritten werden. Wenn wir Zeiger verwenden, dann nehmen wir hier also eine Adressrechnung vor. Wenn in unserem Beispiel

```
int feld [100], *iptr;
iptr = feld;
```

der Inhalt der Zeigervariablen also auf den Feldanfang zeigt, so kann mit `iptr++;` der Zeiger, also die Adresse, inkrementiert werden. Wohin zeigt dann die Variable? Die inkrementierte Variable zeigt nun auf das nächste Element des Feldes, auch dies kann natürlich in einer Schleife erfolgen. Wieder ist die Schleife rechtzeitig zu beenden, um eine Feldüberschreitung zu vermeiden.

Wir haben bis jetzt nur mit einem Integerfeld gearbeitet, was aber wenn man mit einem Character-Feld oder einem Gleitkommafeld arbeitet? Zunächst brauchen wir jetzt eine passende Zeigervariable:

```
char zeichen [50], *cptr;
cptr = zeichen;
double werte [200], *dptr;
dptr = werte;
```

Hier funktioniert die Adressrechnung völlig identisch wie mit dem Integerzeiger. Wenn der Character-Pointer `cptr++` inkrementiert wird, so wird die Adresse um die Größe einer `char`-Zelle (1 Byte) modifiziert; wenn der `double`-Pointer `dptr++` inkrementiert wird,

so wird um die Länge dieser Gleitkommazelle weitergeschaltet, also in der Regel um 8 Byte. Die Adressrechnung arbeitet also abhängig vom Typ des Zeigers jeweils richtig, der Benutzer muss sich darum nicht kümmern. Nun sollte es auch klar sein, warum man mit verschiedenen, jeweils zum Datentyp passenden Zeigertyp arbeiten muss.

`ptr++` bzw. `++ptr`	Zeiger inkrementieren
`ptr--` bzw. `--ptr`	Zeiger dekrementieren
`ptr + n` bzw. `ptr - n`	Zeiger um `n` Elemente erhöhen / erniedrigen; `n` muss Integer sein!
`ptr1 - ptr2`	Differenz von Pointern liefert die Anzahl der Elemente (nicht Bytes) zwischen den Pointern
`ptr1 >= ptr2`	nur sinnvoll, wenn beide Zeiger auf die Elemente eines Vektors zeigen.
nicht erlaubt:	
`ptr1 + ptr2`	sinnlos, ebenso Multiplikation oder Division

Tabelle 11.1: Operationen mit Zeigervariablen

11.2.7 Arithmetik mit Zeigervariablen

Mit Zeigervariablen kann also gerechnet werden, man kann Zeigervariable auch zuweisen, aber hier findet natürlich keine Typwandlung statt wie mit arithmetischen Elementen. Tabelle 11.1 listet die zulässigen Operationen mit Zeigervariablen auf.

Abbildung 11.7 zeigt darüber hinaus einige Beispiele zur Vertiefung.

Rechnen	mit Variablen	mit Adressen
	`int i;`	`int a[N], *pa;`
	`i = 0;`	`pa = a;`
	`i = i+1;`	`pa = pa +1;`
	`i++;`	`pa++;`

Erhöhung	des Wertes	der Adresse
	`k = *pa + 1;`	
	`k = (*pa) + 1;`	`k = *(pa+1);`
	`*pa += 1;`	

Abbildung 11.7: Beispiel für Zeigerarithmetik

Was bedeutet der Ausdruck `*pa++;`?

`(*pa)++; oder *(pa+1); ?`

der Ausdruck `*pa++;` ist identisch mit `*(pa+1);`, es wird die Adresse inkrementiert, also auf das nächste Element im Feld zugegriffen. Soll der Inhalt der Zelle, auf die `pa` zeigt, erhöht werden, so sind Klammern nötig:

```
(*pa)++;
```

Nochmal der Zugriff auf die Elemente eines Feldes `int a[10], *pa;`:

```
a[i] == *(a+i)
a[i] == *(pa+i) mit pa=a
&a[i] == a + i
```

Hat man zwei Vektoren und die zugehörigen Zeiger

```
int a [10] , b [10], *pia, *pib;
pia = a; pib = b;
```

so kann man über den Index oder den Zeiger auf die Feldelemente zugreifen. Zeigervariablen können modifiziert werden (`pia++; pib--;`), nicht aber die Adressen der Felder, `a++;` ist nicht erlaubt. `a` ist zwar auch ein Zeiger – der Zeiger auf das erste Element des Feldes (`[0]`), `a` ist ein *constant pointer*, der nicht modifiziert werden kann. Damit ist auch klar, dass die Zuweisung `a = b;` schön wäre um das gesamte Feld zuzuweisen, aber eben nicht möglich ist. Man kann also nur in einer Schleife Element für Element transferieren, z.B. `*pia = *pib;`. Dies kann man natürlich auch in einer Funktion erledigen.

11.2.8 Priorität von Operatoren (Nachtrag)

Zurück zur Frage, was eigentlich `*pa++` bedeutet. Oben wurden bereits zwei Möglichkeiten erwogen:

a) `(*pa)++` – die Variable, auf die `pa` zeigt, erhöhen, oder

b) `*(pa++)` – die Adresse in `pa` merken, die Adresse dann um eins erhöhen und in `pa` zurückschreiben und schließlich den Wert an der vorher gemerkten Adresse holen.

Für diese Überlegung müssen wir uns zunächst einmal Gedanken darüber machen, welche Priorität die Zeigeroperatoren `&` (Adressoperator) und `*` (Dereferenzierungsoperator) haben. Wir haben bereits in Kapitel 5.4 erfahren, dass diese Operatoren zu den unären (unary) Operatoren gehören, die sich nur auf einen Operanden beziehen. Wir können also die Tabelle der Prioritäten von Operatoren um diese ergänzen:

Primary Operatoren	`()`
Unäre Operatoren	`- + & * ! ++ --`
Multiplikationsoperatoren	`* / %`
Additionsoperatoren	`+ -`
Shiftoperatoren	`<< >>`
Vergleichsoperatoren	`< <= > >=`
Gleichheitsoperatoren	`== !=`
Bitweises UND	`&`
Bitweises XOR	`^`
Bitweises ODER	`|`
Logisches UND	`&&`
Logisches ODER	`||`
Zuweisungsoperatoren	`=`

Tabelle 11.2: Operatorprioritäten (erweitert)

Beachten Sie dabei unbedingt, dass die Operatoren `&` und `*` jeweils als unäre und binäre Operatoren mit verschiedener Bedeutung zur Verfügung stehen.

Bei genauerer Betrachtung der Tabelle fällt nun auf, dass der Dereferenzierungsoperator `*` und der Inkrementierungsoperator `++` auf der selben Prioritätsstufe stehen. Wenn Operatoren der gleichen Priorität gruppiert werden, gibt es zwei Regeln:

- *group right to left* gilt für unary Operatoren und `?`,
- *group left to right* für alle anderen.

Die Lösung für unser Beispiel heißt also:

Beide Operatoren (`*` und `++`) sind unäre Operatoren und somit von gleicher Rangordnung. Es gilt also *group right to left*:

```
*pa++ == *(pa++)
```

– der Zeiger wird erhöht!

Ein Beispiel für binäre Operatoren ist der Ausdruck

```
a = b/c%d;
```

Hier gilt *group left to right* und damit

```
b/c%d == (b/c)%d
```

11.2.9 Vektoren an Funktionen übergeben

C kennt keine Manipulation von zusammengesetzten Objecten, z.B. keine Zuweisung von oder an Vektoren, so kann auch ein Vektor nicht als Ganzes an eine Funktion übergeben werden.

Abhilfe: Der Zeiger auf den Vektor wird übergeben.

Beispiel: Funktion, die das Maximum eines Feldes ermittelt

```
int max ( int *ii )
{
/*
  Die Implementierung
  von max() ist hier
  nicht relevant.
*/
}
int main()
{
  int ivec[10];
  m = max(ivec);
  /* gleichbedeutend mit m= max(&ivec[0]); */
}
```

Der Vektorname ohne Index ist ein Zeiger auf das erste Element (Index: `[0]`)!

Die Funktion `max()` bekommt also nur einen Integer-Pointer, sie kann ihren Parameter als `int`-Vektor oder als „Zeiger auf int“ vereinbaren:

```
max ( int *ii )     oder:    max ( int ii[10] )
{                            {
  ...                          ...
}                            }
```

Bei eindimensionalen Vektoren braucht die Funktion nicht zu wissen, wieviele Elemente der Vektor enthält; legal (und üblich!) ist:

```
max ( int ii[ ] )
{
  ...
}
```

Was ist damit mit der Länge von übergebenen Vektoren? Weder Compiler noch Linker überprüfen, ob die an Funktionen übergebenen Vektoren die richtige Anzahl von Elementen haben (in Pascal ist dies z.B. anderweitig geregelt!).

- Vorteil : es sind Funktionen möglich, die Vektoren beliebiger Länge als Argumente akzeptieren.
- Problem : woher weiß die Funktion die Vektorlänge?
  - durch zusätzlichen Parameter
  - durch spezielles Ende-Element
- Beispiel : Skalarprodukt

Die Länge der Vektoren wird durch zusätzlichen Parameter übergeben:

```
     double scalprod( double *a , double *b , int n )
bzw: double scalprod( double a[ ], double b[ ], int n )
{
  double sum;
  sum = 0.0;

  while (n > 0)
  {
    sum += *a * *b;
    a++;
    b++;
    n--;
  }

  return (sum);
}
```

oder kürzer:

```
double scalprod (double *a , double *b , int n)
{
  double sum;
  for(sum = 0.0; n > 0; n--)
    sum += *a++ * *b++;
  return (sum);
}
```

11.2.10 Strings

Wir wollen noch einmal auf Strings zurückkommen. Strings sind in C kein eigener Datentyp sondern ein *Vektor vom Typ* `char`.

Das letzte Zeichen dieses Character-Feldes ist das Bitmuster 0. Die String-Konstante `"Dies ist ein String"` wird also im Speicher abgelegt als `Dies ist ein String\0` in einem `char`-Vektor mit 20 Elementen, dieser String hat eine konstante Adresse.

```
    char string1[20],
         string2[100];       /* String-Var. = char-Vektoren */
    char *strptr1, *strptr2; /* String-Pointer = Pointer auf char */
    strptr1 = "Hallo Welt";  /* strptr1 zeigt auf String */

damit:
    printf("Hallo Welt\n");  /* direkte Textausgabe */
==  printf("%s\n","Hallo Welt"); /* Ausgabe des Strings "..." */
==  printf("%s\n",strptr1);  /* Ausg. d. Str. via Pointer */

OK:
   strptr2 = strptr1;        /* Zuweisung von Zeigern */
   strptr2 = string1;        /* Adresse von string1 */

nicht OK:
   string1 = "Hallo Welt";
   string1 = strptr1;
```

Die letzten beiden Zuweisungen sind nicht zulässig, da in C keine Verbundanweisung vorhanden ist, d.h. ein Vektor nicht zugewiesen werden kann. Dies gilt natürlich analog für den `char`-Vektor `String`. Es gelten hier dieselben Regeln wie bei der Bearbeitung von Vektoren allgemein. `string1` ist keine Zeigervariable, sondern ein konstanter Zeiger, die Adresse des Vektors im Speicher.

Um einen String (`"..."` oder in einem `char`-Vektor) in ein `char`-Feld umzukopieren, muss jedes Element (incl. `\0`!) einzeln transferiert werden.

- Schleife:

```
char string[20];         /* String-Variable */
char *strptrv, *strptrc; /* String-Pointer */

strptrc = "Hallo Welt"   /* Adr. d. Stringkonstante */
strptrv = string;        /* Adr. d. Stringvektors */
for( i=0; i<10; i++)     /* 10 Zeichen + \0 */
  string[i] = *strptrc++;
  bzw *strptrv++ = *strptrc++;
```

Das kann auch in eine Funktion gepackt werden. Funktionen zum Kopieren von Strings:

- Mit Vektoren:

```
void cop_str_vec(char s1[], char s2[])
{
  int i;
  i = 0;
  while ( s1[i] != 0 ) /* Test auf Stringende */
  {
    s2[i] = s1[i];
    i++;
  }
  s2[i] = \0;
}
```

- Mit Pointern:

```
void cop_str_ptr(char *s1, char *s2)
{
  /* Test auf Stringende, gleichzeitig kopieren */
  while ( (*s2=*s1) != 0 )
  {
    s1++; /* \0 wird mit übertragen */
    s2++;
  }
}
```

- Kurzform:

```
void cop_str_ptr(char *s1, char *s2)
{
  while (*s2++ = *s1++) ; /* Test, Kopie, Inkrement */
}
```

In ANSI-C sind dafür Library-Funktionen vorhanden.
In *string.h* ist deklariert:

```
char *strcpy( char *s1, const char *s2 ); /*Returnwert ist ein char-Pointer auf s1*/
```

Im Beispiel

```
#include <string.h>

char string[20];
strcpy(string, "Hallo Welt");
```

wird der String `"Hallo Welt"` in das Feld `string` kopiert.

Strukturen haben wir schon kennen gelernt; mit Strukturen können, ebenso wie mit den Grunddatentypen, Felder deklariert werden. Hier gelten die gleichen Regeln wie für alle Datentypen. Der Typ `struct datum` sei bekannt:

```
struct datum geb_tage[100], *sptr;
sptr = geb_tage;
sptr++;
```

Zu beachten ist, dass der inkrementierte Zeiger nun nicht auf die nächste Komponente in der Struktur, sondern auf die nächste Struktur im Feld zeigt. Der Zeiger – die Adresse – wird also um die Größe der Struktur weitergeschaltet (Adressrechnung). Der Zugriff auf die Komponenten erfolgt auf die bekannte Weise:

```
sptr -> tag bzw. (*sptr).tag
```

Achtung! In `(*sptr).tag` ist die Klammer erforderlich! Warum?

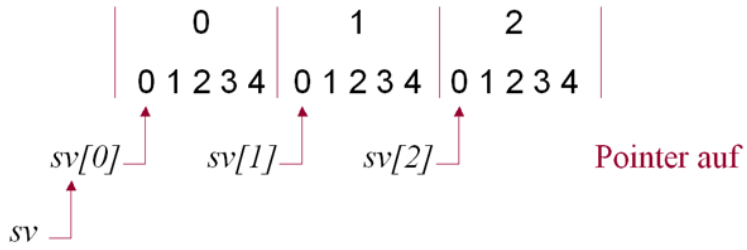
Bei der Übergabe von Strukturen an Funktionen sollte wie auch bei Vektoren immer mit Zeigern gearbeitet werden, es werden also nicht die ggf. umfangreichen Daten übergeben, sondern nur die Adresse, also der Zeiger.

11.2.11 Mehrdimensionale Vektoren (nur für Interessierte)

Komplizierter wird es mit mehrdimensionalen Vektoren, jetzt haben wir einen Vektor von Vektoren z.B.:

```
char sv[3][5] (nicht sv[3,5] !)
\/
prim.expr.
\ /
prim.expr.
\  /
prim.expr.
```

also einen Vektor mit 3 Elementen, von dem jedes Element ein Vektor mit 5 Elementen ist. Man beachte die Reihenfolge, von links nach rechts gelesen. In dieser Reihenfolge liegen die Elemente im Speicher:



Der Zugriff auf die 15 Elemente des 2-dimensionalen Feldes erfolgt über die 2 Indizes, z.B. `sv[1][3]` oder entsprechend über die Zeiger. Nach der Regel „durch Weglassen des Index erhält man einen Zeiger auf das 1. Element“ erhält man hier durch Entfernen eines (des zweiten) Index einen Zeiger auf das 1. Element des entsprechenden Teilfeldes. Lässt man beide Indizes weg, erhält man einen Pointer auf einen Pointer. Mit dem $*$-Operator hat man dann wie bekannt Zugriff auf das Element, auf das der Pointer zeigt. Bei Zeiger auf Zeiger sind demgemäß zwei $**$ erforderlich.

Belegen des ersten Elements:

```
    sv[0][0] = 'a';
    *sv[0] = 'a';
damit:
    (*sv)[0] = 'a';
oder
    *(sv[0]) = 'a';
oder
    printf("%s\n",strptr1); /* Ausg. d. Str. über Pointer */
oder
    **sv = 'a';
    strptr2 = string1; /* Adresse von string1 */
```

Überlegen Sie selbst, auf welches Element zugegriffen wird mit:

```
   sv[0][0] = 'a';

   *sv[1] = 'b';
oder
   (*sv)[1] = 'b';
oder
   *(sv[1]) = 'b';
```

Welche Klammern sind überflüssig ?

```
sv  ist ptr auf --> ptr auf --> (arr[5]) sv[0][0]
                    ptr auf --> sv[1][0]
                    ptr auf --> sv[2][0]
```

`sv[0]`	ist Pointer auf 1.Gruppe (1.Element)
`sv[1]`	ist Pointer auf 2.Gruppe (1.Element)
`sv[2]`	ist Pointer auf 3.Gruppe (1.Element)
damit:	
`*sv[2] == *(sv[2])`	ist 1.Element der 3.Gruppe
aber:	
`(*sv)`	ist Pointer auf Anfang, auf `sv[0]`, nicht auf das 1. Element!
`(*sv)[0]`	ist 1.Element
`(*sv)[1]`	ist 2.Element
:	:
`(*sv)[4]`	ist 5.Element
und	
`**sv`	ist 1.Element
falsch ist:	
`sv = 'x';`	Pointer auf Pointer!
`sv[0]='x';`	Pointer!
`*sv ='x';`	Pointer!

falsch ist auch:

```
sv++ oder *sv++
```

da `sv` oder `*sv` zwar Pointer sind, aber konstante Pointer, es ist die Adresse des Feldes `sv`, die nicht verändert werden kann und darf, richtig wäre:

```
    char *ptrc;
    ptrc = *sv;
dann:
    ptrc++;
```

`ptrc` ist eine Variable, die verändert werden kann.

Zwei- oder auch mehrdimensionale Felder kann man immer auch ausgehend vom ersten Element als eindimensionales Feld betrachten. Beachten Sie dabei die Lage im Speicher, so ist z.B. `*(ptrc + 8)` identisch mit `sv[1][3]`.

11.2.12 Feld von Zeigern

Ebenso wie man Felder mit den verschiedenen Datentypen anlegen kann, kann man auch einen Vektor von Zeigern deklarieren:

```
char feld[20]; /* Feld mit 20 Elementen Typ Zeichen */
char *argv[3]; /* Feld mit 3 Elementen, je Zeiger auf char */
```

Hat `feld` z.B. den Inhalt `"Dies ist ein Text..."`, wie kann dieser Inhalt ins Feld `argv` gebracht werden ?

Der Inhalt des Feldes `argv` ist zunächst noch undefiniert.

Mit

```
argv[0] = &feld[0];
```

zeigt das 1. Element nun auf das 1.Zeichen in `feld` (`D`).

```
argv[1] = &feld[13];
```

zeigt nun auf das Zeichen 'T',

```
*argv[1]='t';
```

verändert dieses Zeichen. `argv` ist ein Zeiger auf den ersten Zeiger, `**argv` ist das erste Zeichen.

Wir haben hier ein eindimensionales Feld von Zeigern, auf das man mit Index oder mit Pointern zugreifen kann. Auch hier arbeitet man dann mit Zeiger auf Zeiger, ähnlich wie bei den mehrdimensionalen Feldern.

Damit:

```
printf("%c",**argv);  /* Erstes Zeichen ==> D */
printf("%s",argv[1]); /* String auf den argv[1] zeigt */
printf("%s",*argv);   /* Erster String */
```

11.2.13 Dynamische Felder

Alle diese Felder (mehrdimensional) oder Vektoren (eindimensional) können statisch mit der entsprechenden Deklaration angelegt werden. Will man das Feld erst während der Laufzeit dynamisch anlegen, so muss man nicht unbedingt mit verketteten Listen arbeiten, man kann sich auch zur Laufzeit den erforderlichen Platz für ein Datenfeld reservieren, z.B.:

```
int *iptr ;
struct datum *sptr ;

/* Platz für 100 int-Werte */
iptr = malloc (100 * sizeof(int)) ;

/* Platz für 20 Strukturen */
sptr = malloc (20 * sizeof(struct datum));
```

Der Zugriff erfolgt nur über den Zeiger, es gibt keine Variablen- oder Feldnamen. Auf den passenden Zeigertyp ist zu achten.

11.3 Input/Output (nur für Interessierte)

I/O ist in C nicht Bestandteil der Sprache, sondern wird über Library-Routinen abgewickelt. Um die Routinen zu verwenden, müssen sie als `extern` deklariert werden. Diese Deklarationen stehen in der Headerfile , die mit `#include` in das Programm eingefügt wird.

11.3.1 Terminal I/O

Die Funktionen `printf()` und `scanf()` greifen auf die vordefinierten „Files“ *stdout* bzw. *stdin* zu, die mit dem Bildschirm, bzw. der Tastatur verknüpft sind. Ebenso greifen die Funktionen `getchar()` und `putchar()` zur Zeichen Ein/Ausgabe, `gets()` und `puts()` für String Ein/Ausgabe auf *stdin* und *stdout* zu. Siehe auch Kapitel 4.

```
/* stdio.h Auszug Terminal I/O */

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

char *gets(char *str);
int puts(const char *str );

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

Achtung: Die Eingabe von Zeichen von der Tastatur erfolgt in der Regel nicht völlig nach dieser Beschreibung, da die Betriebssysteme eine direkte Übergabe an das Programm verhindern, d.h. die Eingabe erst nach <CR> an das Programm erfolgt. Das bedeutet, dass kein Einzelzeichen eingelesen werden kann, da immer ein <CR> eingegeben werden muss. <CR> ist ebenfalls ein Zeichen, das dann auch eingelesen werden muss. Dieses Problem entsteht nur bei Einlesen von Zeichen, bei der Eingabe von Zahlenwerten (`scanf`) wird der <CR> automatisch verworfen. Die in manchen Büchern angegebenen Beispiele funktionieren daher meist nicht. Für die direkte Eingabe einzelner Zeichen können meist unterschiedliche Funktionen (nicht Standard) verwendet werden.

11.3.2 Dateizugriff

Bisher haben wir nur die beiden vordefinierten Kanäle *stdin* (Tastatur) und *stdout* (Bildschirm) verwendet. Zum Einlesen oder Ausgeben von größeren Datenmengen will man aber auch auf Dateien zugreifen. Für den Zugriff auf Dateien muss zuerst eine Verbindung zu einer Datei hergestellt werden. Die Funktion `fopen()`, ebenfalls in *stdio.h* zu finden, liefert einen Pointer auf `FILE`. `FILE` ist ein implementationsabhängiger Typ, meist

eine Struktur, in der die relevanten Informationen für den Filezugriff abgelegt ist. Da dieser Datentyp, die Struktur, bekannt sein muss, wird sie in *stdio.h* definiert.

```
/* STDIO - UNIX 'Standard I/O' Definitions Auszug File I/O */

typedef struct
{
  int _cnt;
  unsigned char *_ptr;
  unsigned char *_base;
  unsigned int _flag;
  unsigned char _fileL;
  unsigned char _fileH;
} FILE ;

#define NULL 0
#define EOF (-1)

FILE * fopen (const char *filename, const char *a_mode);
int fclose (FILE *stream);
```

Die Argumente von `fopen()` sind 2 Strings, also `char`-Pointer, der Filename und die Zugriffsart.

```
FILE *fopen (const char *filename, const char *a_mode, ...);
```

Die Zugriffsart ist:

- `"r"` : lesen (read)
- `"w"` : schreiben (write)
- `"a"` : anhängen (append)

Zu beachten ist, dass für die Zugriffsart auch ein String (`char`-Pointer) anzugeben ist, auch wenn es sich nur um ein Zeichen handelt. Man erhält also einen `FILE`-Pointer, damit kann man auf diese Datei zugreifen. Was sich hinter diesem Pointer verbirgt (in der Regel eine Struktur), braucht den Benutzer nicht zu interessieren. Im Fehlerfall, wenn z.B. die Datei nicht existiert oder die Zugriffsrechte den Zugang verbieten, wird der in *stdio.h* definierte `NULL`-Pointer zurückgeliefert.

Beispiel:

```
if( (fp=fopen("brief.txt","r")) == NULL)
{
```

```
    printf("Open geht nicht");
    /* Programm abbrechen */
    exit(1);
  }
```

Jetzt ist ein Zugriff auf die File möglich, z.B. kann mit

```
fscanf (FILE *file_ptr, const char *format_spec, ...);
```

identisch wie mit `scanf()` gelesen werden, mit

```
int fprintf (FILE *file_ptr, const char *format_spec, ...);
```

identisch wie mit `printf()` in die Datei geschrieben werden.
Für die Datei-I/O wird also hier nur noch das Ziel, bzw. die Quelle (ein `FILE`-Pointer) angegeben. `fprintf` ist also die allgemeine Form für die formatierte Ausgabe.

```
fprintf(stdout, ... ); /* identisch mit printf(...); */
fscanf(stdin, ...);    /* identisch mit scanf(...); */
```

Beispiel:

```
/* Textdatei lesen und auf Terminal ausgeben */
#include <stdio.h>

int main()
{
  FILE *infile;   /* Zeiger auf FILE */
  char wort[100]; /* Text-Buffer */
  infile = fopen("brief.txt","r");
  while ((fscanf(infile, "%s", wort) != EOF)
  {
    printf("%s ", wort);
  }
  fclose (infile);
}
```

`fscanf()` liefert als return-Wert am Ende der Datei `EOF` (-1).
Mit

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

wird die Datei wieder geschlossen, dabei werden die internen Puffer geleert und die Resourcen wieder frei gegeben.

Da `fscanf()` wie `scanf()` nicht zwischen Leerzeichen und Zeilenvorschub unterscheiden kann (beides „white space“) ist das oben geschriebene Programm nicht einwandfrei. Besser sollte mit `getc()` und `putc()` Zeichen für Zeichen gelesen, bzw. geschrieben werden. In *stdio.h* werden die ensprechenden Funktionen deklariert:

```
int fgetc (FILE *file_ptr);
int fputc (int character, FILE *file_ptr);
```

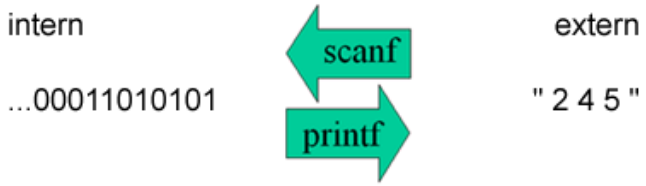
Das verbesserte Textkopier-Programm sieht dann wie folgt aus; in dem Beispiel werden auch die Parameter der Kommandozeile zur Eingabe der Dateinamen verwendet.

```
/* Kopierprogramm copy datei1 datei2 */

#include <stdio.h>

int main(int argc , char *argv[])
{
  FILE *infile, *outfile; /* Zeiger auf FILE */
  int c; /* Buffer für ein Zeichen */

  if (argc != 3)
  {
    fprintf(stderr, "%s: 2 Dateinamen! \n", argv[0]);
    exit(1);
  }

  if((infile = fopen(argv[1],"r")) == NULL)
  {
    fprintf(stderr,"%s: Fehler bei open In-File %s \n",
            argv[0], argv[1]);
    exit(1);
  }

  if((outfile = fopen(argv[2],"w")) == NULL)
  {
    fprintf(stderr,"%s: Fehler bei open Output %s \n",
            argv[0], argv[2]);
    exit(1);
  }
```

```
  while ((c=getc(infile)) != EOF)
  {
    putc(c,outfile);
  }

  fclose (infile);
  fclose (outfile);
}
```

In dem Beispiel wurde zur Ausgabe der Fehlermeldungen der Kanal *stderr* verwendet. *stderr* ist wie *stdin* und *stdout* ein vordefinierter `FILE`-Pointer für die Ausgabe von Fehlermeldungen auf dem Terminal. Damit ist eine saubere Trennung zwischen normaler Programmausgabe (*stdout*) und Fehlermeldungen (*stderr*) möglich. Wird der Ausgabekanal in eine Datei umgelenkt, so werden die Fehlermeldungen weiterhin auf dem Bildschirm ausgegeben.

Bei der formatierten Ein-/Ausgabe wird immer das interne Bitmuster in eine darstellbare Zeichenfolge (bzw. umgekehrt) gewandelt:



also Umwandlung von Zahlenwerten in einen String entsprechend dem Format.

Strings können aber auch in einem Character-Feld abgelegt werden. Als Ziel oder Quelle für einen formatierten I/O kann ein Zeichen-Feld angegeben werden.

```
int sprintf(char *string, const char *format, ...);
int sscanf(char *string, const char *format, ...);
```

Die Funktionen `sprintf()` und `sscanf()` arbeiten analog zu `printf()`, `scanf()`, nur dass die Ein/Ausgabe von bzw. in einen String erfolgt.

Beispiel:

```
char s[10];
int i=245, j=678;

sprintf(s, "%d %d", i, j);

/*
   |---|---|---|---|---|---|---|---|---|
   |'2'|'4'|'5'|' '|'6'|'7'|'8'| 0 |
   |---|---|---|---|---|---|---|---|---|
*/
```

Der Inhalt des Feldes `s`, ein String, kann ausgegeben werden:

```
printf("%s" , s);
```

oder wieder eingelesen werden:

```
sscanf(s, "%d ", &j);
```

damit wird der 1.Wert (245) in die Zelle `j` eingelesen.

11.4 Argumente der Kommandozeile (nur für Interessierte)

Der Start eines Programms erfolgt unter dem Betriebssystem UNIX durch Eingabe des Dateinamen (ohne Extension), also der Datei in der das lauffähige Programm abgelegt ist. Bei diesem Programmaufruf können in der Kommandozeile noch weitere Eingaben für das Programm erfolgen. Dies kennen Sie von verschiedenen Utility-Programmen. In einem C-Programm stehen die beim Aufruf in der Kommandozeile angegebenen Argumente als Strings zur Verfügung.

Jedes C-Hauptprogramm beginnt mit `int main()`. Das Programm ist eine Funktion mit dem speziellen Namen `main`, diese Funktion wird beim Programmstart vom Betriebssystem aufgerufen. Wie bei einer Funktion können beim Aufruf Parameter übergeben werden, in diesem Fall die in der Kommandozeile zusätzlich angegebenen Argumente. Zu diesem Zweck ist `main` wie eine Funktion (des Betriebssystems) deklariert:

```
int main(int argc, char *argv[]);
```

also mit einem Integer-Parameter und einem Feld von Character-Zeigern. Dabei ist:

`argc`	Anzahl der Argumente
`argv[]`	Liste mit Pointern auf die Argument-Strings

Wenn ein Programm in der Form

```
name arg1 arg2 arg3 ...
```

aufgerufen wird, so stehen im Programm die Anzahl der Argumente, sowie in

`argv[0]`	Pointer auf den Namen des Programms,
`argv[1]`	Pointer auf das 1. Argument `arg1` (String),
⋮	⋮
`argv[argc-1]`	Pointer auf des letzte Argument.

zur Verfügung.

Beispiel:

```
copy brief.txt text.txt
```

Die Strings `"brief.txt"` und `"text.txt"` sind Argumente für das Programm *copy*.

Der Zugriff im Programm auf diese Kommandozeile sieht wie folgt aus:

```
int main(int argc, char *argv[])
{
  printf("Anzahl: %d",argc); /* ==> Anzahl: 3 */
  printf("%s", argv[0]);     /* ==> copy      */
  printf("%s", argv[1]);     /* ==> brief.txt */
  printf("%s", argv[2]);     /* ==> text.txt  */
}
```

oder allgemein:

```
for (i=1; i < argc; i++)
  printf(("%s ", argv[i]); /* ==> alle Argumente */
```

bzw.

```
while (--argc > 0)
  printf("%s ", *++argv); /* ==> alle */
```

Wird in diesen Beispielen der Programmname mit ausgegeben? Damit kann eine Bearbeitung der Eingabeparameter erfolgen. Zu beachten ist, dass es sich dabei immer um Textelemente handelt (Zeiger auf Character); sollen also in der Kommandozeile auch Zahlenwerte eingegeben werden, so müssen diese erst entsprechend umgewandelt werden.

Wie kann auf das erste Zeichen des 5. Parameters Zugegriffen werden? Hierzu gibt es verschiedene gleichwertige Möglichkeiten:

```
argv[5][0]  (*(argv+5))[0]  *argv[5]  **(argv+5)
```

Beim Aufruf des *cc*-Compilers werden üblicherweise die Optionen als Parameter mit einem -Zeichen beginnend eingegeben, suchen Sie nach diesem Zeichen.

```
int main(int anz, char *par[])
{
  if (par[1][0] == '-') ...   /* 1.Zeichen 1. Parameter     */
  if ((*++par)[0] == '-') ... /* 1. Zeichen, alle Parameter */
}
```

Untersuchen Sie, ob und warum alle Klammern erforderlich sind.

12 Das Modulkonzept, externe Module

12.1 Module

Ein Programm besteht in der Regel aus mehreren Modulen, d.h. ein Programm (Hauptmodul) kann mehrere Module enthalten und aus externen Modulen oder aus Bibliotheken Dinge wie Funktionen, Konstanten, Variablen oder Datentypen verwenden. Dieses Konzept führt zu einer Gliederung des Programms, bei der das Programm in kleinere Einheiten zerlegt wird, die einzeln und unabhängig voneinander entwickelt und getestet werden können.

Das Modulkonzept unterstützt das Entwurfsprinzip der schrittweisen Verfeinerung (Abstraktion und Konkretisierung), wobei die Funktion des Moduls zunächst auf einer hohen Abstraktionsebene beschrieben wird, um sie dann allmählich durch Ausführung der einzelnen Algorithmen bis ins Detail zu konkretisieren.

C erzwingt zwar kein strenges Modulkonzept, aber eine Aufspaltung des Programmes in einzelne Module wird ermöglicht. In C gibt es die Möglichkeit, Module separat vom Hauptprogramm-Modul zu erstellen und zu übersetzen. Diese Module werden dann wie auch Bibliotheks-Module zum Programm dazugebunden (hinzugelinkt).

12.2 Schnittstellen

Leider gibt es keine strenge Definition der Schnittstelle, sondern nur die besprochene Deklaration von Funktionen und Variablen. Wenn man sich aber an das bei den vorhandenen Bibliotheks-Modulen verwendete Prinzip hält, dass diese Vereinbarungen in einem Headerfile zusammengestellt werden, so ist eine übersichtliche Aufspaltung in einzelne Module möglich und sinnvoll.

Solche Module bestehen also aus dem eigentlichen Funktionscode und einer dazugehörigen Headerdatei. Der Quellcode wird separat übersetzt, ausgetestet und dann zu einem Programm-Modul dazugebunden. Das spart Zeit beim Compilieren eines Programms, weil das externe Modul nicht mitübersetzt werden muss. Änderungen an einem externen Modul können völlig getrennt durchgeführt werden. Nach der Übersetzung des Moduls muss das Programm nur neu gelinkt werden.

Solange die Schnittstelle gleich bleibt, können einzelne Module ausgetauscht werden, ohne Änderungen am Hauptprogramm vornehmen zu müssen! „Schnittstelle“ meint hier alle Funktionsdeklarationen, Typendefinitionen und globalen Variablen, die vom Haupt-

modul (oder auch einem anderen Modul) benutzt werden. Diese Schnittstelle ist im Header enthalten, die eigentliche Implementierung in der entsprechenden C-Datei.

12.3 Beispiel

In unserem Beispiel stehe also in einer Datei *mittelwert.c* (Implementierung):

```
/* Berechnung des Mittelwerts eines Gleitkommafeldes */

float mittelwert(float arr[], int laenge)
{
  float sum=0.0;
  int i;
  for (i=0; i<laenge; i++)
    sum=sum+arr[i];
  return sum/laenge;
}
```

Eine weitere Datei, das Headerfile *mittelwert.h*, enthält die Deklaration der Funktion(en) (Schnittstelle):

```
/* Berechnung des Mittelwerts eines Gleitkommafeldes */

float mittelwert(float arr[], int laenge);
```

Ein Programm, das diese Funktion verwendet, muss dieses Headerfile einfügen:

```
#include "mittelwert.h"
```

Hier wird die zweite Form der `#include`-Anweisung verwendet. Die Datei *mittelwert.h* ist im lokalen Bereich des Benutzers zu finden, der Name wird deshalb in Anführungszeichen gesetzt.

Die Headerfiles der Bibliotheksfunktionen stehen systemweit zur Verfügung, der Dateiname steht daher in spitzen Klammern, wie z.B.:

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

Das Hauptprogramm steht in einer eigenen Datei *beisp.c*:

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

int main()
```

```
{
  float feld[100], a[50], b[500];
  float x,y,z;
  ...
  z = mittelwert (feld, 100);
  ...
}
```

Das Hauptprogramm (*beisp.c*) und das Modul (*mittelwert.c*) werden getrennt übersetzt. Es entstehen die Dateien *beisp.o* und *mittelwert.o*. Beim Linken von *beisp* würde nun aber die Implementierung der Funktion `mittelwert` fehlen – man muss dem Linker also „sagen“, wo sich diese Funktion befindet. Wir wissen ja, dass sie sich in der Datei *mittelwert.o* befindet, und müssen dies dem Linker mitteilen (je nach Compiler/Linker über andere Einstellungsmöglichkeiten).

12.4 Header

Im Header sollten immer nur die Deklarationen stehen, nie direkt C-Code. Sinnvoll und notwendig kann es aber sein, die Vereinbarung neuer Datentypen (Aufzählungstyp, Strukturen. . .) in den Header zu schreiben, wenn sie „außen“ (also von den aufrufenden Programmteilen) mitbenutzt werden.

Warum sollen im Header kein Code und keine globalen Variablen stehen? Der Header wird ja meist in mehrere Implementierungsfiles (*.c*) eingebunden (`#include`). Dieses Einbinden findet auf rein textuelle Art statt: an der Stelle, wo im Quelltext das `#include` steht, wird (vom Präprozessor) direkt der Header eingefügt (per Textersetzung). Erst danach wird diese Datei an den Compiler übergeben.

In einem Headerfile selbst können auch weitere `#include`s stehen. Diese weiteren Dateien werden dann einfach auch mit eingebunden.

12.5 `extern` und Linker

Wenn ein Programm nun aus mehreren Teilen besteht, die ja getrennt übersetzt werden, weiß der Compiler natürlich nicht mehr, was aus einem Header stammt und was aus einem C-File kommt. Entsprechend würde der Code aus dem Header in jedem Binärfile enthalten sein, dessen C-File den Header eingebunden hatte. Das ist nicht nur Verschwendung, sondern macht auch beim Linken Probleme: wann soll (gerade bei globalen Variablen) denn nun welche Version benutzt werden?

Bei Funktionen kann man diese Probleme ja immer vermeiden, indem man einfach keine Funktionen in Header schreibt. Was aber, wenn man (warum auch immer) unbedingt

globale Variablen verwenden will/muss und diese dann auch noch in unterschiedlichen Modulen zur Verfügung stehen sollen?

Auch hierfür gibt es in C (natürlich) eine Lösung: das Schlüsselwort `extern`. Mit diesem Schlüsselwort wird dem Compiler mitgeteilt, dass sich eine Variable nicht in der aktuellen Datei befindet, sondern eine Variable dieses Namens aus einem anderen Modul gemeint ist.

Sie sollten aber, soweit möglich, auf globale Variablen verzichten. Verwenden Sie auch in den Implementierungsdateien (z.B. *mittelwert.c*) soweit irgendwie möglich keine globalen Variablen. Vereinbart man nämlich woanders zufällig eine globale Variable gleichen Namens, kommt es zu einer Fehlermeldung des Compilers und an verschiedenen Stellen des Programms müssen Variablennamen wieder geändert werden.

Große Programme bestehen aus vielen einzelnen Modulen, die von mehreren Programmierern entwickelt werden. Verwendet jeder großzügig globale Variablen, so muss man am Ende – eigentlich völlig unnötig – noch einmal viel Zeit in Nachbesserungen stecken, wenn es zu Namenskonflikten kommt.

12.6 Zusammenführen („Linken“) von Modulen

Das Übersetzen von Programmen, die aus mehreren Modulen bestehen, gliedert sich in mehrere Schritte. Zunächst muss jedes Modul einzeln kompiliert werden (sofern dies noch nicht geschehen ist). Dabei erhält man sogenannte Objektdateien, die auf *.o* enden. Diese Dateien müssen anschließend zu einer ausführbaren Datei zusammengefasst werden. Diesen Schritt bezeichnet man als *Linken*. Da das Linken ein unabhängiger Vorgang ist, können beim Linken natürlich auch Fehler auftreten. Sollten sie also beim Übersetzen Fehlermeldungen erhalten, müssen sie unterscheiden, ob diese vom Compiler oder vom Linker stammen.

A Fehlersuche

Die Suche von Fehlern in Software nimmt häufig wesentlich mehr Zeit in Anspruch als das eigentliche Schreiben der Programme. Lassen Sie sich davon nicht entmutigen, dieses Problem haben auch erfahrene Programmierer. Je genauer man aber seine Programme plant, bevor man mit dem eigentlichen Schreiben beginnt, desto weniger muss man dann später mit dem Suchen von Fehlern verbringen. Es lohnt sich also sehr wohl und spart auch insgesamt Zeit, sich zunächst Gedanken zur Lösung zu machen, und dann erst mit dem Schreiben zu beginnen.

A.1 Programmentwurf

A.1.1 Struktogramme

Um den Ablauf eines Programms besser zu verstehen, helfen bereits einfache Struktogramme (z.B. Nassi-Shneiderman-Diagramme). Skizzieren Sie also am besten Ihr Programm erst auf einem Blatt Papier, bevor Sie mit dem Implementieren beginnen.

A.1.2 Formatierung

Eng verbunden mit dem Planen von Software und der Darstellung als Diagramme ist das übersichtliche Formatieren des Quelltextes. Es ist schon verblüffend, wie viel undurchschaubarer Code geschrieben wird. Allein durch passendes Einrücken zusammengehörender Programmblöcke gewinnt ein Programm ungemein an Übersichtlichkeit. Genau diese Übersicht ist aber Grundvoraussetzung für das Verständnis eines Programms, gerade dann, wenn es nicht das macht, was der Programmierer erwartet.

Achten Sie speziell darauf, dass die geschweiften Klammern eines Anweisungsblocks in der gleichen Spalte sind, und die Anweisungen innerhalb des Blocks weiter eingerückt sind als die umschließenden Klammern.

```
/* sehr schlechter Stil: */

for (x=1; x<10; x++) {
for (y=1; y=5; y++) {
printf("%d\n",y); }
printf("%d\n",x);
}
```

```
/* immer noch unübersichtlich und verwirrend: */

for (x=1; x<10; x++) {
  for (y=1; y=5; y++) {
    printf("%d\n",y); }
    printf("%d\n",x);
}


/* wesentlich besser: */
for (x=1; x<10; x++)
{
  for (y=1; y=5; y++)
  {
    printf("%d\n",y);
  }
  printf("%d\n",x);
}
```

Wie man sieht, deutet allein das Einrücken eine Zusammengehörigkeit von Anweisungen an. Wenn die Einrückungen nun nicht den tatsächlichen Anweisungsblöcken entsprechen, wird das Lesen und Verstehen des Programms völlig unnötig erschwert.

```
/* "böse" Falle: */

for (x=1; x<10; x++);
{
  printf("%d\n",x);
}
```

Das letzte Beispiel zeigt eine etwas versteckte Fehlerquelle: es sieht zwar so aus, als ob in der `for`-Schleife jeweils der aktuelle Wert von `x` ausgegeben wird. Nur steht nach der Schleife ein Strichpunkt (`;`), und damit eine leere Anweisung. Der Anweisungsblock wird daher auch nur einmal ausgeführt, und zwar nach dem Ende der Schleife.

A.1.3 Zwischenwerte

Sehr hilfreich ist es auch, sich Gedanken über die erwarteten Zwischenergebnisse zu machen. Diese „Soll“-Werte kann man dann gut mit den tatsächlich berechneten „Ist“-Werten vergleichen. Dieser Vergleich von Wunsch und Realität kann dann sehr gezielt zur Fehlersuche eingesetzt werden.

Die gewünschten (Soll-) Werte (z.B. für Zwischenergebnisse) sollte der Programmierer im Vorfeld ermitteln (anhand der Aufgabenstellung). Woher aber weiß man nun, welche (Ist-) Werte eine Variable tatsächlich annimmt?

Man gibt diese Variable einfach per `printf()` aus. Solche Test-Ausgaben kann man später ja wieder entfernen, wenn das Programm fehlerfrei läuft. Diese sehr einfache Vorgehensweise genügt meist, um auch hartnäckige Fehler aufzuspüren. Gute Kandidaten für die Ausgabe sind neben berechneten Zwischenwerten auch Schleifenzähler und Variablen, die als Index für ein Feld benutzt werden.

A.1.4 Debugger

Die Suche nach Fehlern in einem Programm nennt man im Englischen „Debugging“, also das Beheben von „Bugs“, d.h. Softwarefehlern. Debugger sind Programme, die diesen Prozess vereinfachen, indem sie ermöglichen das fehlerhafte Programm Befehl für Befehl abzuarbeiten und dabei jederzeit den aktuellen Wert von Variablen, bzw. den Inhalt des Speichers auszugeben.

A.2 Fehlermeldungen und Warnungen des Compilers

Fehlermeldungen und Warnungen sind zwei unterschiedliche Dinge: Bei Fehlermeldungen wird das Programm nicht übersetzt, während bei Warnungen der Compiler trotzdem „weitermacht“. Während Fehlermeldungen also gar nicht ignoriert werden können, ist dies bei Warnungen durchaus möglich.

In der Praxis sieht man nun zwei Effekte: zum einen werden vom Compiler schon bei kleinsten Programmierfehlern Unmengen an Fehlermeldungen und/oder Warnungen ausgegeben, die teilweise keinerlei ersichtlichen Grund haben und ohne sehr detaillierte Kenntnisse auch völlig unverständlich bleiben. Zum anderen werden, gerade wegen ihrer schieren Menge, Warnungen gerne ignoriert („Hauptsache das Programm läuft irgendwie, dann sind die Warnungen doch egal“).

Man kann aber auch ohne großen Aufwand sehr leicht Programme schreiben, die ohne eine einzige Warnung übersetzt werden. Manche Warnungen sind tatsächlich „überflüssig“, das liegt daran, dass der Compiler den Code nur übersetzt, ihn aber eben nicht wirklich versteht. Er kann also nur verdächtig aussehende Konstrukte anmahnen, auch wenn diese vielleicht funktionieren und evtl. sogar so beabsichtigt sind. Sehr häufig weisen diese Warnungen aber tatsächlich auf versteckte Fehler hin, die sich vielleicht nur in ganz ungewöhnlichen Situationen auch auswirken, und dann sucht man den Fehler um so länger. Es hilft also sehr wohl auch dem Programmierer, die Warnungen zu beachten und deren Ursache zu beheben.

Was aber macht man nun mit der Unmenge an Fehlermeldungen bzw. Warnungen? Gerade wenn sie völlig sinnlos erscheinen? Der wichtigste Tip hier ist: zunächst immer nur den ersten Fehler bzw. die erste Warnung betrachten. Alles andere sind meist Fol-

gefehler, die durch das Beheben des ersten Fehlers auch verschwinden. Also: ersten Fehler suchen, korrigieren und dann neu übersetzen. Werden immer noch Fehlermeldungen ausgegeben, dann betrachtet man wieder zunächst die erste Meldung.

Da ein Compiler die Absicht des Programmierers ja nicht kennen kann, findet er natürlich nur einen Teil der Fehler, und bemerkt diese häufig auch erst an einer späteren Stelle. Für Sie als Programmierer bedeutet dies, dass eine Fehlermeldung (auch die erste!) durchaus von einem Fehler herrühren kann, der einige Zeilen vor der beanstandeten Stelle liegt.

A.3 Abstürze (z.B. segmentation violation)

Programmabstürze werden i.A. durch falsche Speicherzugriffe ausgelöst. Im Skript wurde ja bereits an vielen Stellen auf diese Problematik hingewiesen (speziell bei Feldern und Zeigern). Wie findet man aber nun die Ursache?

Bei Feldern empfiehlt es sich, vor jedem Zugriff auf ein Element den Index, mit dem auf das Feld zugegriffen wird, per `printf()` auszugeben. Diese Ausgabe vergleicht man dann mit der tatsächlichen Größe des Feldes: wurde z.B. ein Feld `a[10]` definiert, so darf bekanntlich nur auf die Elemente 0 bis 9 zugegriffen werden.

Bei Zeigern wird die Sache schon schwieriger. Hier kann es aber bereits helfen, einen Überblick über den Ablauf des Programms zu haben und zu wissen, wann bzw. wo es abstürzt. Man fügt dazu per `printf()` an „interessanten“ Stellen eine Ausgabe ein (jeweils mit unterschiedlichem Text!). Gute Stellen sind z.B. vor und nach Operationen mit Zeigern und die Zweige von `if`-Blöcken. Das Ziel ist, mit Hilfe der tatsächlich erfolgten Ausgaben den tatsächlichen Ablauf mitverfolgen zu können. Das genügt häufig schon, um zu verstehen, wo etwas schief läuft.

B Referenzlisten

B.1 ASCII-Tabelle

00 00	NUL	01 01	SOH	02 02	STX	03 03	ETX	04 04	EOT	05 05	ENQ	06 06	ACK
07 07	BEL	08 08	BS	09 09	HT	10 0A	NL	11 0B	VT	12 0C	NP	13 0D	CR
14 0E	SO	15 0F	SI	16 10	DLE	17 11	DCL	18 12	DC2	19 13	DC3	20 14	DC4
21 15	NAK	22 16	SYN	23 17	ETB	24 18	CAN	25 19	EM	26 1A	SUB	27 1B	ESC
28 1C	FS	29 1D	GS	30 1E	RS	31 1F	US	32 20	SP	33 21	!	34 22	"
35 23	#	36 24	$	37 25	%	38 26	&	39 27	’	40 28	(	41 29	)
42 2A	*	43 2B	+	44 2C	,	45 2D	-	46 2E	.	47 2F	/	48 30	0
49 31	1	50 32	2	51 33	3	52 34	4	53 35	5	54 36	6	55 37	7
56 38	8	57 39	9	58 3A	:	59 3B	;	60 3C	<	61 3D	=	62 3E	>
63 3F	?	64 40	@	65 41	A	66 42	B	67 43	C	68 44	D	69 45	E
70 46	F	71 47	G	72 48	H	73 49	I	74 4A	J	75 4B	K	76 4C	L
77 4D	M	78 4E	N	79 4F	O	80 50	P	81 51	Q	82 52	R	83 53	S
84 54	T	85 55	U	86 56	V	87 57	W	88 58	X	89 59	Y	90 5A	Z
91 5B	[	92 5C	\	93 5D	]	94 5E	^	95 5F	-	96 60	‘	97 61	a
98 62	b	99 63	c	100 64	d	101 65	e	102 66	f	103 67	g	104 68	h
105 69	i	106 6A	j	107 6B	k	108 6C	l	109 6D	m	110 6E	n	111 6F	o
112 70	p	113 71	q	114 72	r	115 73	s	116 74	t	117 75	u	118 76	v
119 77	w	120 78	x	121 79	y	122 7A	z	123 7B	{	124 7C	|	125 7D	}
126 7E	~	127 7F	DEL										

Tabelle B.1: ASCII-Zeichensatz

Die Steuerzeichen von 0 bis 31 (Dezimal) sind in folgender Tabelle nochmal etwas ausführlicher dargestellt. Jedes Zeichen ist mit seiner Nummer in Oktal, Dezimal und Hexa-

dezimal angegeben, zusätzlich das tatsächlich dargestellte Zeichen (z.B. bei Benutzung von *more* oder *nedit*), der Name des Symbols, der ausführlichere Zeichenname, und schließlich die Tastenkombination.

Achtung: einige Tastenkombinationen werden auch von der Shell benutzt und funktionieren daher nicht so wie hier dargestellt!

Okt	Dez	Hex	Darst	Symbol	Zeichenname	Tastatur
000	0	00	^none	NUL	Null	Ctrl-Shift-@
001	1	01	^B	SOH	Start of Header	Ctrl-B
002	2	02	^C	STX	Start of Text	Ctrl-C
003	3	03	^D	ETX	End of Text	Ctrl-D
004	4	04	^E	EOT	End of Transmission	Ctrl-E
005	5	05	^F	ENQ	Enquire	Ctrl-F
006	6	06	^G	ACK	Acknowledge	Ctrl-G
007	7	07	^H	BEL	Bell	Ctrl-H
010	8	08	^I	BS	Back Space	Ctrl-I
011	9	09	^J	HT	Horizontal Tab	Ctrl-J
012	10	0A	^K	LF	Line Feed	Ctrl-K
013	11	0B	^L	VT	Vertical Tab	Ctrl-L
014	12	0C	^M	FF	Form Feed	Ctrl-M
015	13	0D	^N	CR	Carriage Return	Ctrl-N
016	14	0E	^O	SO	Shift Out	Ctrl-O
017	15	0F	^P	SI	Shift In	Ctrl-P
020	16	10	^Q	DLE	(or DEL) Delete	Ctrl-Q
021	17	11	^R	DC1	Device Control 1	Ctrl-R
022	18	12	^S	DC2	Device Control 2	Ctrl-S
023	19	13	^T	DC3	Device Control 3	Ctrl-T
024	20	14	^U	DC4	Device Control 4	Ctrl-U
025	21	15	^V	NAK	Negative Acknowledge	Ctrl-V
026	22	16	^W	SYN	Sychronize	Ctrl-W
027	23	17	^X	ETB	End Transmission Block	Ctrl-X
030	24	18	^Y	CAN	Cancel	Ctrl-Y
031	25	19	^Z	EM	End of Medium	Ctrl-Z
032	26	1A	^[	SUB	Substitute	Ctrl-[
033	27	1B	^	ESC	Escape	Ctrl-
034	28	1C	^]	FS	File Separator	Ctrl-]
035	29	1D	^^	GS	Group Separator	Ctrl-^
036	30	1E	^_	RS	Record Separator	Ctrl-_
037	31	1F	^	US	Unit Separator	Ctrl-

Tabelle B.2: ASCII-Steuerzeichen

B.2 Priorität der Operatoren

Es wurde im Skript mehrfach auf die unterschiedliche Priorität von Operatoren hingewiesen. Hier finden Sie nun die vollständige Liste; sie enthält auch viele Operatoren, die im Rahmen dieser Einführung nicht behandelt wurden.

Zu jedem Operator werden sein englischer Originalname und ein Beispiel angegeben. In den Beispielen werden folgende Platzhalter verwendet:

- *member* (Name eines Elements)

- *object* (Ausdruck, der ein Objekt einer Klasse liefert)
- *pointer* (Ausdruck, der einen Zeiger liefert)
- *expr* (Ausdruck)
- *lvalue* (nicht-konstantes Objekt, also veränderbare Variable)
- *type* (Typname)
- *(type)* (Kompletter Typname, mit *, () usw.)

Unäre Operatoren (mit nur einem Operanden) und Zuweisungs-Operatoren sind rechts-assoziativ, alle anderen links-assoziativ:

- `a=b=c` steht für `a=(b=c)`, weil rechts-assoziativ (Zuweisungs-Operatoren)
- `a+b+c` steht für `(a+b)+c`, weil links-assoziativ
- `*p++` steht für `*(p++)`, weil rechts-assoziativ (zwei unäre Operatorn mit gleicher Priorität)

Jeder Kasten der Tabelle enthält Operatoren gleicher Priorität; die Operatoren eines Kasten haben höhere Priorität als die Operatoren in den nachfolgenden Kästen:

- `a+b*c` steht für `a+(b*c)`, weil `*` eine höhere Priorität besitzt als +
- `a+b-c` steht für `(a+b)-c`, weil beide Operatoren die gleiche Priorität besitzen (und links-assoziativ sind)
- `a & mask == b` steht für `a & (mask == b)` und nicht für `(a & mask) == b`

Gerade das letzte Beispiel zeigt, dass man nicht intuitiv erfassbare Ausdrücke besser mit Klammern versieht.

Name	**Operator**
Indizierung	`zeiger[ausdruck]`
Funktionsaufruf	`ausdruck(ausdrucksliste)`
Elementzugriff	`objekt.element`
Dereferenzierender Elementzugriff	`zeiger->element`
Postinkrement	`lvalue++`
Postdekrement	`lvalue--`
Werterzeugung	`(typ){ausdrucksliste}`

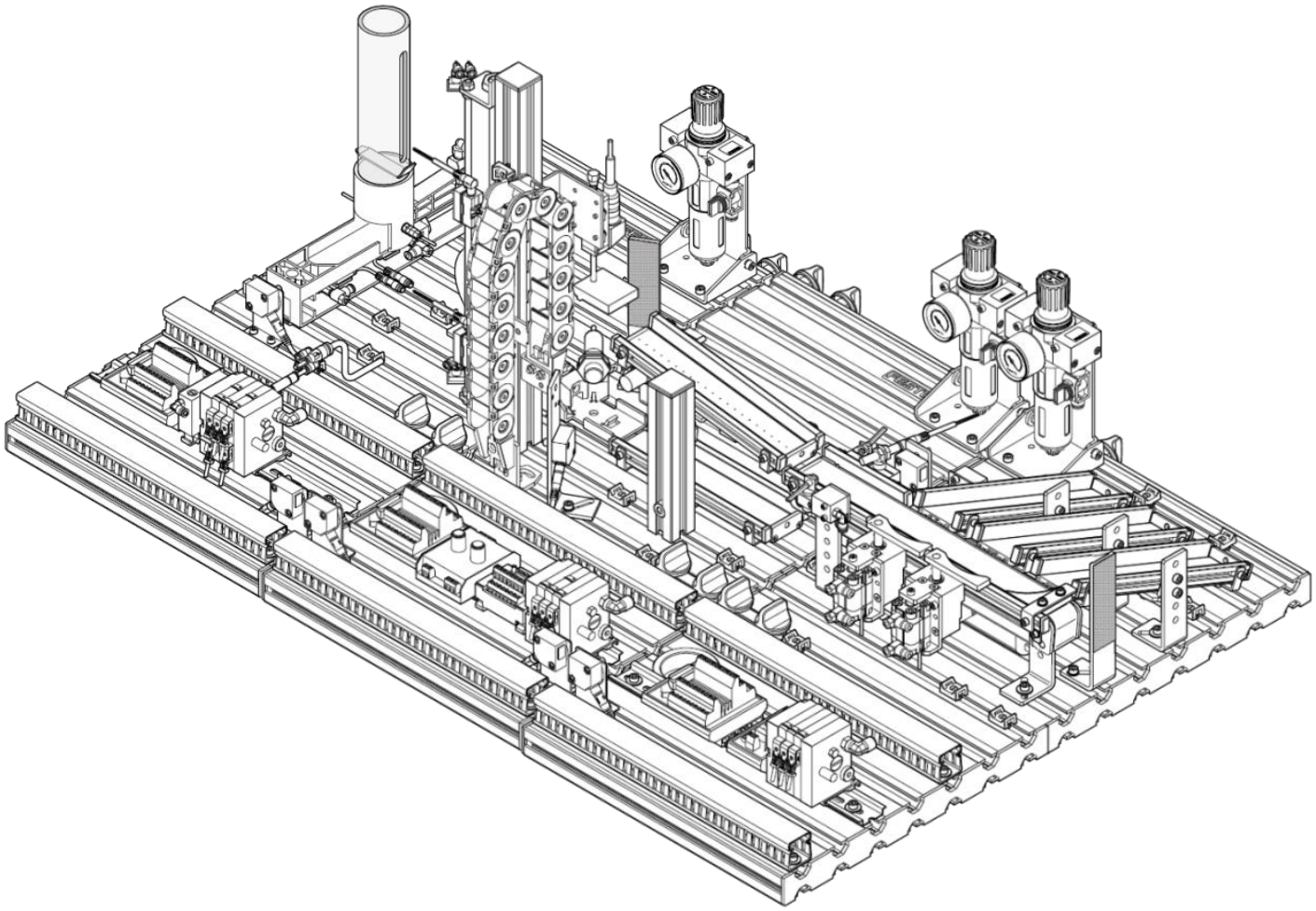
Präinkrement	`++lvalue`
Prädekrement	`--lvalue`
Adresse	`&lvalue`
Dereferenzierung	`*ausdruck`
Einstelliges Plus	`+ausdruck`
Einstelliges Minus	`-ausdruck`
Komplement	`~ausdruck`
Nicht	`!ausdruck`
Objektgröße	`sizeof ausdruck`
Typgröße	`sizeof (typ)`
Typkonversion	`(typ) ausdruck`
Multiplikation	`ausdruck * ausdruck`
Division	`ausdruck / ausdruck`
Modulo (Rest)	`ausdruck % ausdruck`
Addition	`ausdruck + ausdruck`
Subtraktion	`ausdruck - ausdruck`
Linksschieben	`ausdruck << ausdruck`
Rechtsschieben	`ausdruck >> ausdruck`
Kleiner als	`ausdruck < ausdruck`
Größer als	`ausdruck > ausdruck`
Kleiner gleich	`ausdruck <= ausdruck`
Größer gleich	`ausdruck >= ausdruck`
Gleich	`ausdruck == ausdruck`
Ungleich	`ausdruck != ausdruck`
Bitweises Und	`ausdruck & ausdruck`
Bitweises Exklusiv-Oder	`ausdruck ^ ausdruck`
Bitweises Oder	`ausdruck | ausdruck`
Logisches Und	`ausdruck && ausdruck`
Logisches Oder	`ausdruck || ausdruck`
Bedingte Zuweisung	`ausdruck ? ausdruck : ausdruck`
Zuweisung	`lvalue = ausdruck`
Multiplikation und Zuweisung	`lvalue *= ausdruck`
Division und Zuweisung	`lvalue /= ausdruck`
Modulo und Zuweisung	`lvalue %= ausdruck`
Addition und Zuweisung	`lvalue += ausdruck`
Subtraktion und Zuweisung	`lvalue -= ausdruck`
Linksschieben und Zuweisung	`lvalue <<= ausdruck`
Rechtsschieben und Zuweisung	`lvalue >>= ausdruck`
Und und Zuweisung	`lvalue &= ausdruck`
Exklusiv-Oder und Zuweisung	`lvalue ^= ausdruck`

Oder und Zuweisung	`lvalue |= ausdruck`
Sequenz	`ausdruck , ausdruck`

Tabelle B.3: Jede Tabellenzeile enthält Operatoren gleicher Priorität. Operatoren in den unteren Tabellenzeilen haben niedrigere Priorität als Operatoren in den oberen Tabellenzeilen. Einige Regeln der Syntax lassen sich aber nicht durch diese Tabelle beschreiben.

Anlagen-Übungsskript

Grundlagen der modernen Informationstechnik 2



Stationsbeschreibung & Aufgabenstellung

Einleitung

Dieses Dokument soll Ihnen helfen sich auf das Anlagenpraktikum im Rahmen der Vorlesung „Grundlagen der modernen Informationstechnik 2“ vorzubereiten. Daneben soll es möglichst viele Fragen und Unsicherheiten bezüglich des Anlagenpraktikums im Voraus klären.

Dieses Skript ist ausschließlich zur Vorbereitung auf das Testat und die Prüfung vorgesehen und darf im Anlagenpraktikum nicht verwendet werden! Dokumentationen werden für jeden Teilnehmer gestellt.

Ablauf Anlagenpraktikum

Begeben Sie sich rechtzeitig, je nachdem für welche Gruppe Sie sich angemeldet haben, in den richtigen Raum („Technikraum“: MW 0102; „CIP“: MW U050). Die Sitzplätze werden durch die Tutoren nach dem Zufallsprinzip zugeteilt.

Das Anlagenpraktikum besteht aus den folgenden Teilen:

Praktikumsteil	Dauer	zum Bestehen nötige Punkte	maximal erreichbare Punkte
Eingangstestat	0:10 Stunden	4	10
Zustandsdiagramm	ca. 0:30 Stunden	-	-
Anlagen-programmierung	ca. 3:00 Stunden	0	18 (Einzelanlage) + 5 (Teambonus)
Gesamt	3:45 Stunden	-	28 + 5

Zunächst wird über das zehn Minuten lange *Eingangstestat* festgestellt, ob eine ausreichende Vorbereitung auf das Anlagenpraktikum durch den Studenten durchgeführt wurde. Dafür müssen im Testat mindestens vier der zehn möglichen Punkte erreicht werden. Das Testat wird im Anschluss ausgewertet und den Teilnehmern, die das Testat nicht bestanden haben, innerhalb weniger Minuten Bescheid gegeben.

Falls Sie das Testat mit einer ausreichenden Punktzahl von mindestens vier Punkten bestanden haben, haben Sie im Rahmen der *Anlagenprogrammierung* die Möglichkeit innerhalb der nächsten 3,5 Stunden eine Ihnen zufällig zugewiesene automatisierte Station zu programmieren. Zunächst müssen Sie den Steuerungsablauf auf Papier in Form eines Zustandsdiagramms modellieren. Nach Abnahme durch die Tutoren können Sie dann mit der eigentlichen Programmierung beginnen. Gelingt es Ihnen die zugewiesene Station so zu programmieren, dass der in diesem Dokument festgelegte Testfall funktioniert, kann das Programm unter Vorführung von einem Tutor abgenommen werden. Zusätzlich zu diesen Punkten können fünf weitere Punkte (Überhang) durch Teamarbeit erlangt werden. Dazu muss es Ihnen und den zwei Teilnehmern, welche die beiden angrenzenden Stationen programmieren, gelingen, den korrekten Gesamtablauf der drei Anlagen durchzuführen. Auch hier ist ein festgelegter Testfall einem der Tutoren zu demonstrieren.

Programme können frühzeitig durch Tutoren abgenommen werden. Sie müssen dann nicht die restliche Zeit anwesend sein.

Anlagenpraktikumsregeln

Im Anlagenpraktikum sind einige Regeln zu beachten, welche für eine faire Durchführung einzuhalten sind.

Zum Praktikum dürfen lediglich ***zwei leere DIN A4-Notizblätter*** sowie ***Schreibutensilien*** mitgebracht werden. Für die Programmierung wird Ihnen ein Skript der Ihnen zugewiesenen Station zur Verfügung gestellt. Bei dem Skript handelt es sich um eine gekürzte Version dieses Skripts, welches die Kapitel „Werkstücke“, die entsprechende Stationsbeschreibung und die Sicherheitshinweise beinhaltet. Die während dem Praktikum ***angefertigten Notizen dürfen nicht mitgenommen werden***, sondern müssen im Anschluss bei den Tutoren abgegeben werden.

Beachten Sie, dass es sich bei den Testaten um eine Prüfungssituation handelt und damit die Benutzung von Mobiltelefonen, insbesondere zum Fotografieren und zur Videoaufnahme, nicht gestattet ist.

Anlagenpraktikumsvorbereitung

Zur Vorbereitung auf das Anlagenpraktikum stehen Ihnen verschiedene Übungsmöglichkeiten zur Verfügung.

Zunächst dient die *Anlagenübung* als Einführung in das Thema der Anlagenprogrammierung in dem Rahmen, in dem das Anlagenpraktikum abgehalten wird. Hier werden in zwei Übungen die nötigen Grundlagen der Automatisierungstechnik, die Zerlegung und Modellierung des Ablaufs, sowie das Ausprogrammieren der Abläufe vermittelt.

Zusätzlich wird eine *Heimarbeit* mit vorbereitenden Aufgaben rechtzeitig in Moodle bereitgestellt. In dieser Heimarbeit können Teile der Zentralübung in praktischer Anwendung von den Studenten geübt werden.

Weiterhin finden sie auf Moodle ein *Anlagensimulationsmodell mit Skript*. Das Modell können Sie auf Ihrem eigenen Rechner programmieren und so selbstständig die für das Anlagenpraktikum relevanten Punkte vertiefen.

Das *Übungsskript* dient der selbstständigen Vorbereitung auf die Aufgaben innerhalb des Anlagenpraktikums. Hier sind alle möglichen zu bearbeitenden Aufgaben der Anlagenprogrammierung beschrieben.

Werkstücke

Jede der Stationen bearbeitet sogenannte Werkstücke. Dabei handelt es sich um zylindrische Plastikteile, bei denen sich auf einer Seite eine Öffnung befindet, auf die zusätzlich ein Deckel geschraubt werden kann.

Die Werkstücke haben folgende Farben und Höhen:

Farbe	Höhe	Höhe mit Deckel
Schwarz	22 mm	25 mm
Silber	25 mm	28 mm
Rot	25 mm	28 mm

Silberne Werkstücke besitzen metallische Eigenschaften und können durch Induktionssensoren als solche erkannt werden.

Hinweis

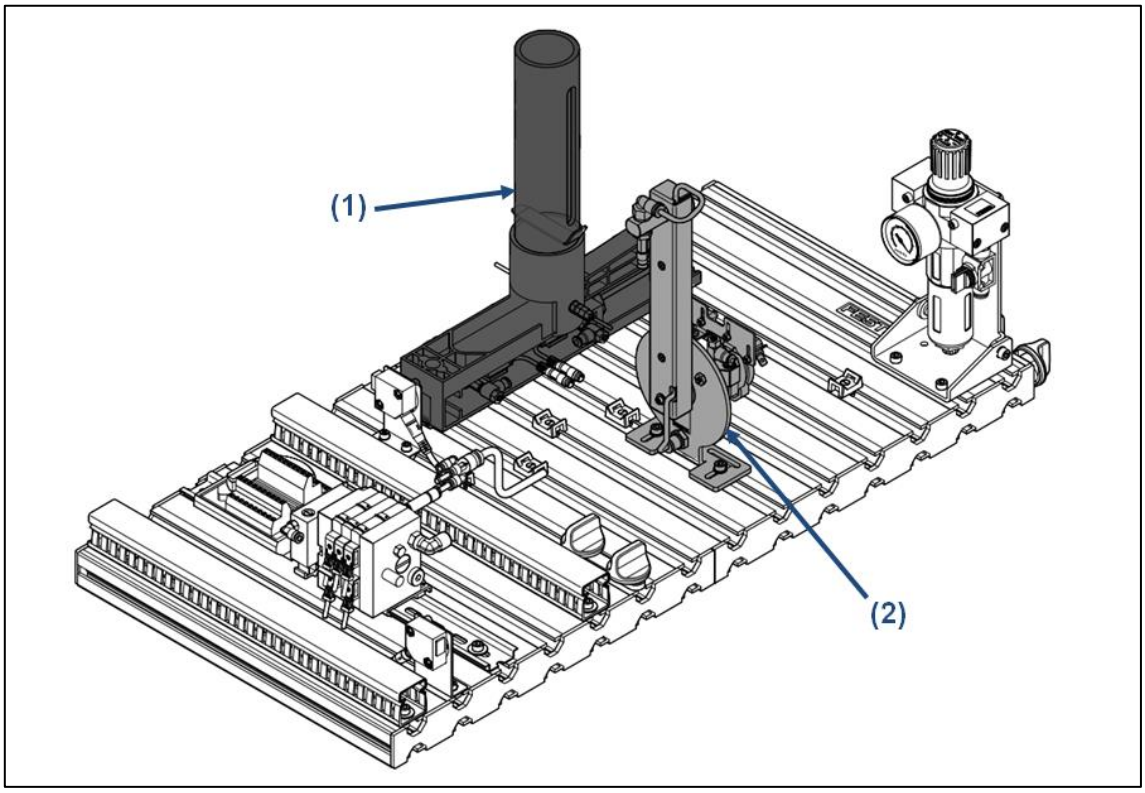
Werkstücke ohne Deckel sind mit Öffnung nach unten in die Stationen einzulegen. Ist ein Deckel auf das Werkstück aufgeschraubt, so ist für die Einlegeorientierung die Anlagenkombination ausschlaggebend. Handelt es sich um die Kombination „Verteilen, Prüfen, Sortieren“, so ist das Werkstück mit Deckel nach oben einzulegen. In der Kombination „Verteilen, Handhaben, Trennen“ ist das Werkstück mit Deckel nach unten einzulegen. Diese Orientierungen sind wichtig, damit die richtige Funktionsweise der Stationen gewährleistet werden kann. Andernfalls kann das Werkstück in der Station Handhaben oder an der Prüfstation beim Umsetzen anecken, da der Deckel geringfügig größer als der Grundkörper ist.

Stationsbeschreibung

Im Folgenden werden die einzelnen Stationen beschrieben, von denen Sie eine innerhalb des Anlagenpraktikums programmieren sollen. Die Anlage wird Ihnen zufällig zugewiesen werden. Die Anlagen sind in zwei verschiedene, jeweils aus drei Stationen bestehende Anlagenkombinationen eingebunden. Diese zwei Kombinationen sind „Verteilen, Prüfen, Sortieren“ und „Verteilen, Handhaben, Trennen“.

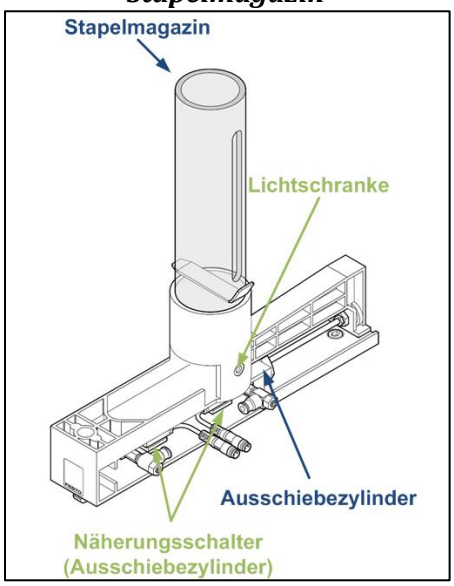
1. Station „Verteilen“

Die Station Verteilen vereinzelt Werkstücke. Im *Stapelmagazin (1)* haben bis zu acht Werkstücke Platz, die über einen einfachwirkenden Zylinder einzeln ausgestoßen werden können. Der *Schwenkarm (2)* greift anschließend die Werkstücke mithilfe eines Vakuumsaugers und befördert sie zur Folgeanlage.



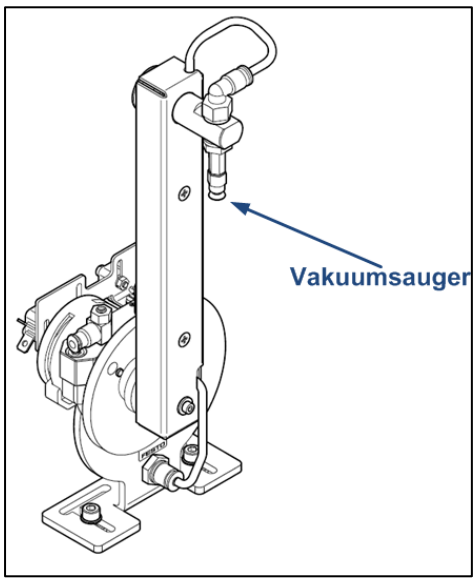
Module

Stapelmagazin



Das Modul dient dazu Werkstücke bis zu ihrer Verwendung zu lagern. Hierfür verfügt das Modul über ein Stapelmagazin, das bis zu acht Werkstücke aufnehmen kann. Das Vorhandensein eines Werkstücks kann über eine Lichtschranke nachgewiesen werden. Um ein Werkstück aus dem Magazin zu schieben, existiert ein selbstrückstellender Ausschiebezylinder, dessen Endlagen über Sensoren abgefragt werden können.

Umsetzer



Das Modul dient dazu die Werkstücke zur Folgestation zu bringen. Hierfür verfügt das Modul über einen pneumatischen Umsetzer, an dessen Ende ein Vakuumsauger installiert ist. Beide Endlagen des Umsetzers lassen sich über Sensoren abfragen, ebenso wie das erfolgreiche Ansaugen eines Werkstücks. Zur Abgabe eines Werkstücks kann außerdem ein Abstoßimpuls gegeben werden.

Ablauf

Zu Anfang wird der Umsetzer zum Magazin gedreht und es wird kein Werkstück angesaugt. Der Ausschiebezylinder ist eingefahren, also an der Position „Magazin“. Wenn der Ausgangszustand hergestellt ist, beginnt die Anlage mit folgender Routine:

Nur wenn ein Werkstück erkannt wird und ein Freigabesignal der Folgestation vorhanden ist, dreht der Umsetzer zur Folgestation, um den Weg für den Ausschiebezylinder frei zu machen. Der Ausschiebezylinder schiebt nun ein Werkstück aus, welches der Umsetzer abholt. Er dreht dazu zurück in Richtung Magazin und schaltet, dort angekommen, nach kurzer Wartezeit das Vakuum ein und fährt den Zylinder wieder ein (Werkstück sonst eingeklemmt). Sobald das Vakuum aufgebaut ist, wird das Freigabesignal der Folgestation geprüft. Besteht Freigabe, dreht der Umsetzer anschließend dorthin und stößt das Werkstück ab. Nachdem das Werkstück abgestoßen wurde, kann ein weiteres Werkstück, falls vorhanden, ausgeschoben und der Ablauf wiederholt werden. Falls nicht, sollte der Umsetzer in den Ausgangszustand am Magazin gebracht werden, um Beschädigungen durch die Folgestation zu vermeiden.

Hinweise

- Legen Sie die Werkstücke ohne Deckel mit Öffnung nach unten in das Stapelmagazin.
- Ist ein Deckel auf das Werkstück aufgeschraubt, legen Sie das Werkstück mit Deckel nach oben in das Stapelmagazin, falls die Folgestation die Station „Prüfen“ ist.
- Ist die Folgestation „Handhaben“ legen Sie das Werkstück mit Deckel nach unten ein.

- Der Umsetzer bewegt sich nicht, falls beide Verfahrsignale gesetzt sind.
- Bei Verfahren des Umsetzers oder des Ausschiebezylinders immer Kollisionssituationen bedenken!
- Beachten Sie, dass das Freigabesignal der Folgestation hardwaregebunden standardmäßig gegeben wird (folgestation_belegt = 0), selbst wenn die Folgestation nicht läuft oder programmiert wird.
- Prüfen Sie das Freigabesignal erst, wenn der Umsetzer wirklich bereit zum Umsetzen ist (Vakuum aufgebaut). Achten Sie auch darauf, dass das Freigabesignal während des Umsetzens zurückgenommen werden kann!

Wichtige Variablen:

Eingänge (Rückmeldung der Anlage)

Bit	Variablenname	Beschreibung
1	sensoren.zylinder_eingezogen	Zylinder im Magazin ist eingezogen (Werkstücke können nachfallen)
2	sensoren.zylinder_ausgeschoben	Zylinder im Magazin ist ausgeschoben
3	sensoren.umsetzer_vakuum	Unterdruck am Sauger des Umsetzers ist aufgebaut
4	sensoren.umsetzer_magazin	Umsetzer an der Abholposition des Stapelmagazins
5	sensoren.umsetzer_folgestation	Umsetzer an der Folgestation
6	sensoren.magazin_leer	Kein Werkstück in Lichtschranke des Fallmagazins
7	sensoren.folgestation_belegt	Folgestation ist nicht für Aufnahme von Werkstücken bereit

Hinweis: Das Freigabesignal kann in dieser Station auch durch den Schlüsselschalter an der Bedienkonsole vorgetäuscht werden, falls die Folgestation beispielsweise nicht programmiert wird oder eine Anlagenabnahme unabhängig von der Folgestation durchgeführt werden soll.

Ausgänge (Befehle von Benutzer an Anlage)

Bit	Variablenname	Beschreibung
0	aktoren.zylinder_ausschieben	Zylinder ausfahren, also Werkstück zur Abholposition schieben. Ist dieser Wert nicht gesetzt, bewegt sich der Zylinder in die Ausgangsstellung zurück (fährt ein).
1	aktoren.umsetzer_vakuum	Unterdruck zum Ansaugen eines Werkstücks anschalten
2	aktoren.umsetzer_abstossen	Werkstück abstoßen
3	aktoren.umsetzer_magazin	Umsetzer zur Abholposition am Magazin bewegen
4	aktoren.umsetzer_folgestation	Umsetzer zur Folgestation bewegen

Abnahme

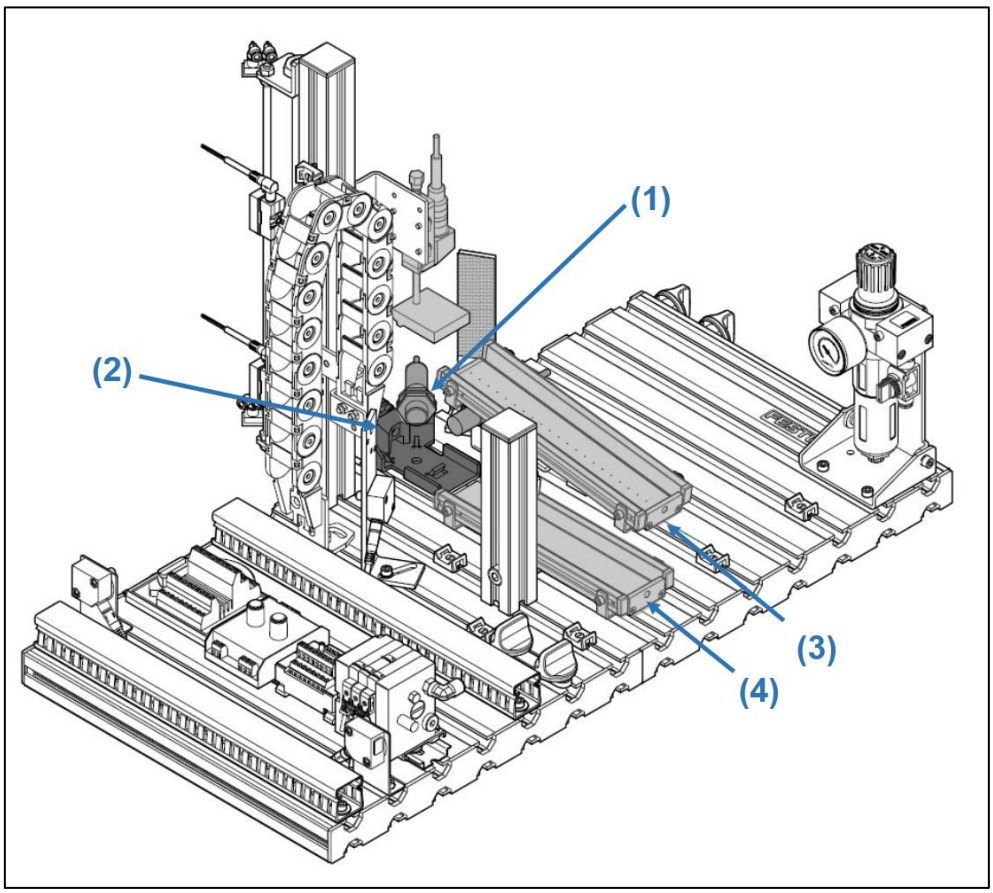
Zur Abnahme der Station sind folgende Eigenschaften zu erfüllen:

- Ausgangssituation laut Ablaufbeschreibung.
- Umsetzer darf sich nur zur Folgestation bewegen, wenn ein Freigabesignal gegeben wurde.
- Umsetzer verbleibt nicht unnötig lange bei Folgestation, sondern bewegt sich unabhängig davon ob das Stapelmagazin voll oder leer ist wieder in die Ausgangsstellung zurück.

Es werden zwei Werkstücke (schwarz ohne Deckel, rot/silbern mit Deckel) in zufälliger Reihenfolge zusammen in das Stapelmagazin gelegt, welche erfolgreich zur Folgestation transportiert werden müssen. Nach dem Transport verfährt der Umsetzer in die Ausgangsposition.

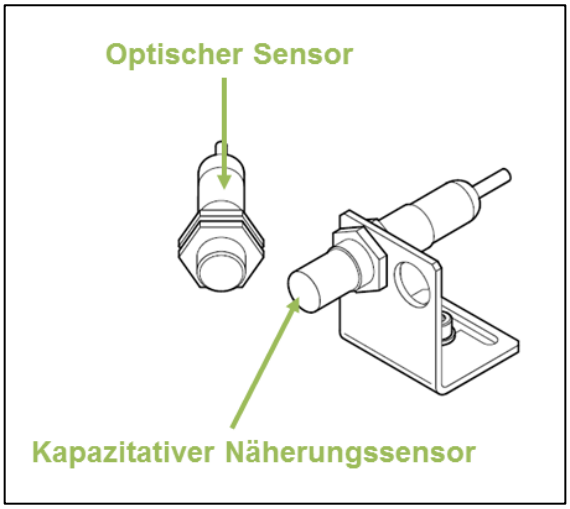
2. Station Prüfen

Die Station „Prüfen“ sortiert Werkstücke ohne Deckel aus. Dafür wird das Werkstück über eine *Hebebühne (2)* angehoben und seine Höhe gemessen. Gleichzeitig wird die Farbe der Werkstücke (schwarz/nicht-schwarz) am *Erkennungsmodul (1)* unterschieden. Abhängig vom Ergebnis wird das Werkstück entweder auf der *Luftkissenrutsche (3)* zur Folgestation geleitet oder von der Hebebühne wieder abgesenkt und auf die *Ausschussrutsche (4)* befördert.



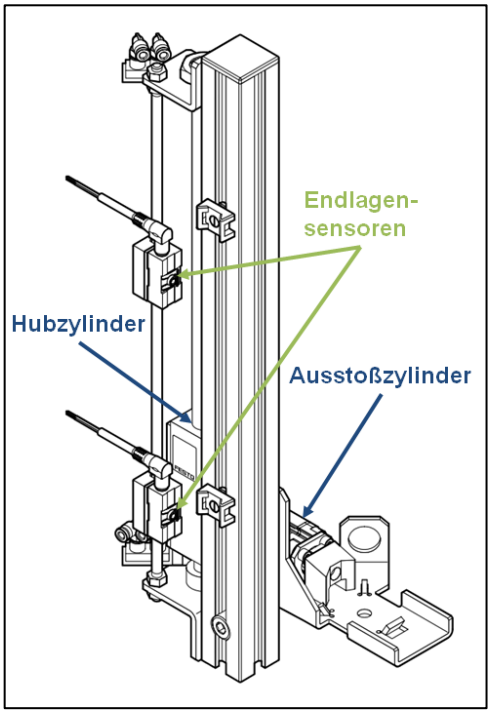
Module

Erkennen



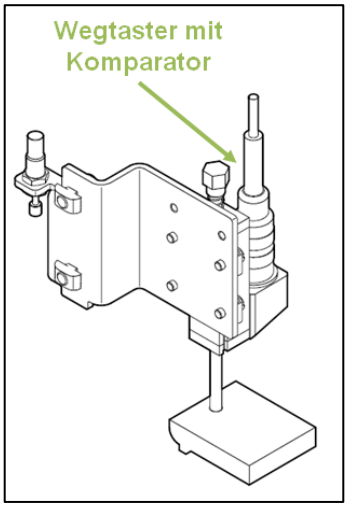
Das Modul dient dazu schwarze von nicht-schwarzen Werkstücken zu unterscheiden. Hierfür ist ein optischer Sensor installiert. Um die Präsenz eines Werkstücks – gleich welcher Farbe – zu prüfen ist außerdem ein kapazitativer Näherungssensor verfügbar.

Hebebühne



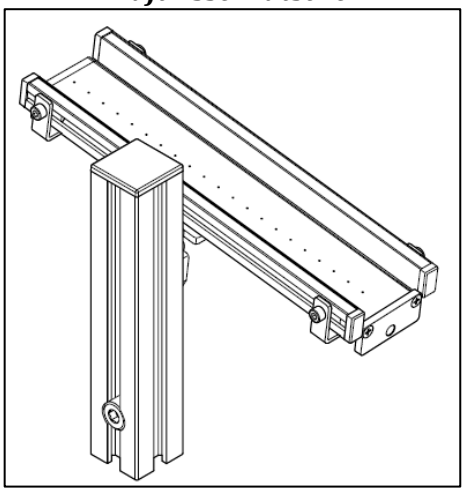
Das Modul dient dazu Werkstücke zur Höhenprüfung zum Modul *Messen* anzuheben, sowie Werkstücke auf eine der beiden Rutschen auszustoßen. Zum Anheben existiert ein Hubzylinder, dessen Endlagen abgefragt werden können. Zum Ausstoßen wird ein selbstrückstellender Ausstoßzylinder verwendet.

Messen



Das Modul kann über einen Wegtaster die Werkstückhöhe erfassen. Durch ein Potentiometer kann ein abzuprüfendes Höhenintervall festgelegt werden, z. B. um festzustellen, ob die Höhe des Werkstücks im richtigen Rahmen liegt.

Luftkissenrutsche



Das Modul befördert bei angeschalteter Druckluft ein Werkstück zur angeschlossenen Nachfolgestation.

Ablauf

Zu Anfang befindet sich der Aufzug an der unteren Position, der Ausstoßzylinder ist eingezogen, es ist dort kein Werkstück vorhanden und die Luftrutsche ist ausgeschaltet. Zudem wird kein Freigabesignal gegeben. Wenn der Ausgangszustand hergestellt ist, beginnt die Anlage mit folgender Routine:

In diesem Zustand (keine Werkstück vorhanden) wird der vorhergehenden Station das Freigabesignal gegeben, welche daraufhin ein Werkstück liefert. Sobald ein Werkstück erkannt wird, wird das Freigabesignal zurückgenommen. Wenn der Arbeitsraum frei ist, wird nach kurzer Wartezeit (ca. 1 Sek) das Werkstück mit Hilfe der Hebebühne angehoben und durch das resultierende Andrücken an das Messmodul auf seine Bauhöhe überprüft. Vor der Höhenmessung muss kurz gewartet werden, um ein Nachschwingen der Feder zu vermeiden. Gleichzeitig kann mit dem optischen Sensor festgestellt werden, ob das Bauteil schwarz ist oder nicht. Werkstücke mit Deckel (*hoehe_ok != werkstueck_nicht_schwarz*) werden, sobald das Freigabesignal der Folgestation vorliegt, an der oberen Hebebühnenposition auf die Luftrutsche (diese zuvor anschalten) ausgestoßen, wonach der Aufzug wieder in seine Ausgangsposition bewegt wird. Handelt es sich dagegen um ein Werkstück ohne Deckel, wird das Werkstück erst auf der unteren Hebebühnenposition auf die Ausschussrutsche ausgestoßen. Befindet sich die Anlage wieder in Ausgangsposition, kann der Ablauf von neuem begonnen werden.

Hinweise

- Bei den Messungen sollte sich das Werkstück eine kurze Zeit (z. B. drei Sekunden) in Ruhe bei dem Sensor befinden, bevor der Wert abgefragt wird.
- Es gibt keinen Sensor mit dem geprüft werden kann, ob der Ausstoßzylinder vollständig ausgefahren ist. Verwenden sie stattdessen eine geeignete Kontrollstruktur.
- Beim Verfahren des Aufzuges muss immer auf einen freien Arbeitsraum geachtet werden!
- Bei Vorgängen deren Endposition nicht durch Endlagensensoren überprüft werden können, müssen Sie zur Annahme des Erreichens der Endlage Zeitmessungen durchführen. Gehen Sie zunächst von ausreichend großen Werten aus (z. B. fünf Sekunden), bevor Sie die Zeiten später optimieren.
- Justieren Sie gegebenenfalls den Komparator zur Festlegung des Prüfintervalls des Höhensensors. Drehknopf 1 (Level 1) legt die untere, Drehknopf 2 (Level 2) die obere Schranke des Intervalls fest. Befindet sich der Messwert im Intervall, so wird eine 1 zurückgegeben. Ist der Sensor richtig eingestellt, sollte er die Werte in untenstehender Tabelle zurückgeben. (Sollten Sie nicht sicher sein, kontaktieren sein einen Tutor, bevor sie Änderungen vornehmen.)
- Ziehen Sie folgende Tabelle zu Rate, um Werkstücke mit Deckel mit den gegebenen Sensoren zu identifizieren:

	Mit Deckel		**Ohne Deckel**	
	hoehe_ok	*hoehe_ok!=*	*hoehe_ok*	*hoehe_ok!=*
	werkstueck_nicht_schwarz	*werkstueck_nicht_schwarz*	*werkstueck_nicht_schwarz*	*werkstueck_nicht_schwarz*
schwarz	1	1	0	0
	0		0	
rot oder metallisch	0	1	1	0
	1		1	

Wichtige Variablen

Eingänge (Rückmeldung der Anlage)

Bit	Variablenname	Beschreibung
0	sensoren.werkstueck_vorhanden	Auf dem Aufzug wurde ein Werkstück erkannt (liefert nur auf unterer Aufzugposition richtige Daten)
1	sensoren.werkstueck_nicht_schwarz	Werkstück auf dem Aufzug ist nicht schwarz (Sensor bewegt sich mit Aufzug mit)
2	sensoren.arbeitsraum_frei	Der Arbeitsraum ist frei, der Umsetzer der Vorgängerstation ist auf Ausgangsposition (am Magazin)
3	sensoren.hoehe_ok	Die gemessene Höhe liegt im eingestellten Intervall (schwarz mit Deckel / rot ohne Deckel / silbern ohne Deckel gibt jeweils eine 1, sonst 0)
4	sensoren.aufzug_oben	Der Aufzug ist oben
5	sensoren.aufzug_unten	Der Aufzug ist unten
6	sensoren.zylinder_eingefahren	Der Ausschiebezylinder ist zurückgezogen (wirft nicht aus)
7	sensoren.folgestation_belegt	Folgestation ist nicht für Aufnahme von Werkstücken bereit

Ausgänge (Befehle von Benutzer an Anlage)

Bit	Variablenname	Beschreibung
0	aktoren.aufzug_abwaerts	Aufzug nach unten fahren
1	aktoren.aufzug_aufwaerts	Aufzug nach oben fahren
2	aktoren.zylinder_ausschieben	Ausschiebezylinder ausfahren, Werkstück auf eine der Rutschen schieben (Zylinder bewegt sich zurück, falls Wert nicht gesetzt)
3	aktoren.luft_an	Luftkissen auf der oberen Rutsche anschalten
7	aktoren.station_frei	Freigabesignal an vorhergehende Station geben

Abnahme

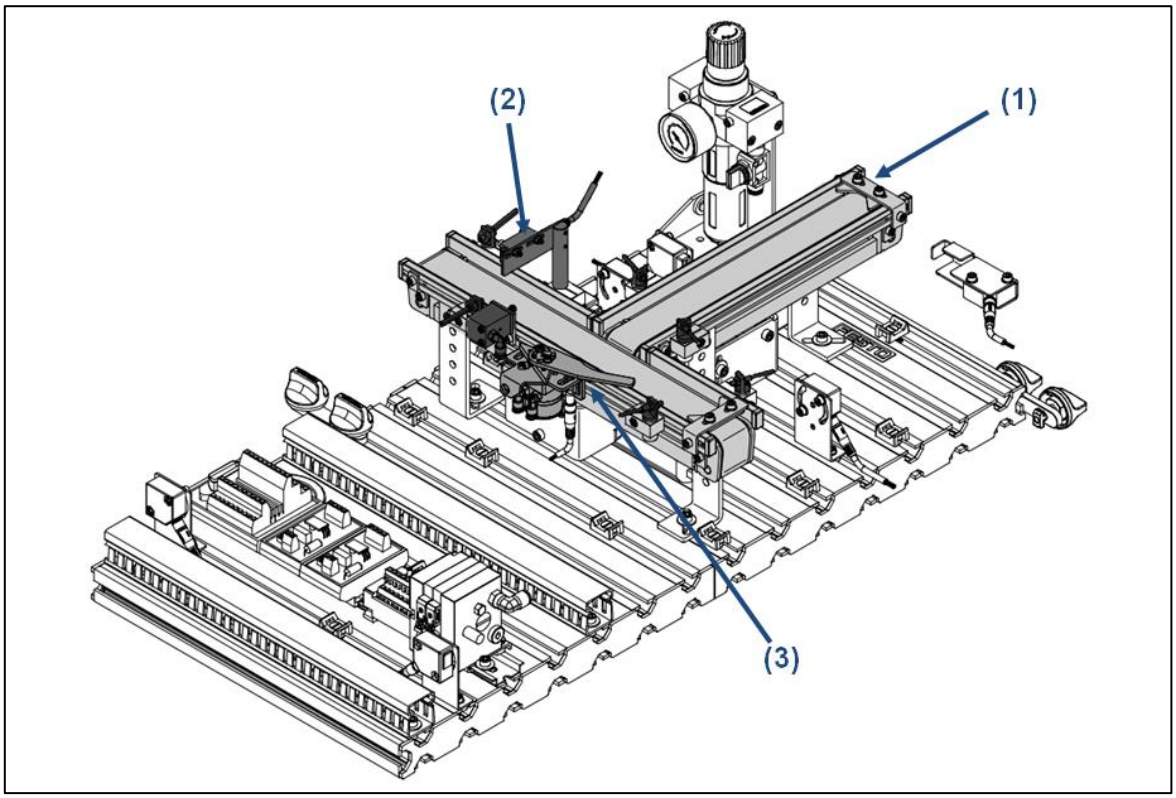
Zur Abnahme der Station sind folgende Eigenschaften zu erfüllen:

- Ausgangszustand laut Ablaufbeschreibung.
- Alle Werkstücke werden so sortiert, dass Werkstücke mit Deckel auf der Luftrutsche weitertransportiert werden und Werkstücke ohne Deckel auf die Ausschussrutsche befördert werden.
- Die Luftrutsche ist nur so lange wie nötig angeschaltet.

Es werden vier Werkstücke (schwarz ohne Deckel, rot/silbern ohne Deckel, rot/silbern mit Deckel, schwarz mit Deckel) in zufälliger Reihenfolge einzeln in die Aufnahme gelegt. Neue Werkstücke werden erst auf die Aufnahme gelegt, wenn das Freigabesignal an die Vorgängerstation gegeben wurde.

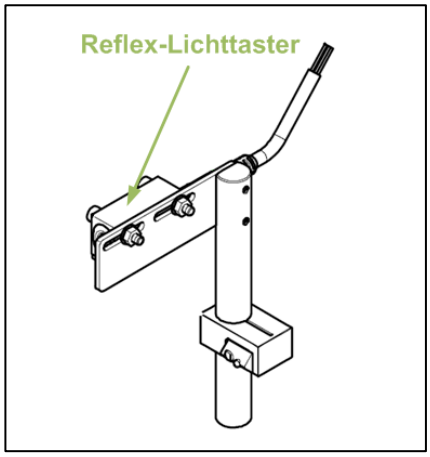
3. Station „Trennen“

Die Station „Trennen“ kann verschiedene Werkstücke unterscheiden und die nicht weiter benötigten auf ein *Ausschussband (1)* auslagern. Die Unterscheidung der Werkstücke erfolgt über eine *Höhenmessung (2)*. Abhängig von der Bewertung befördert eine *Weiche (3)* die Werkstücke auf das Ausschussband.



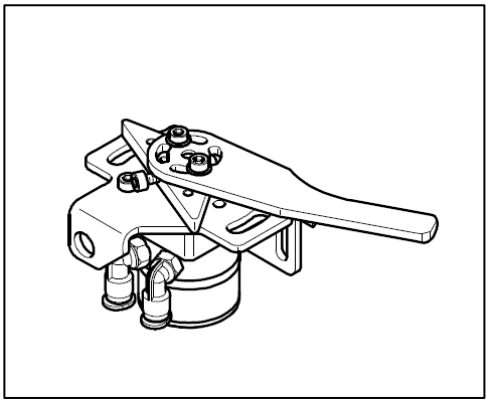
Module

Unterscheiden



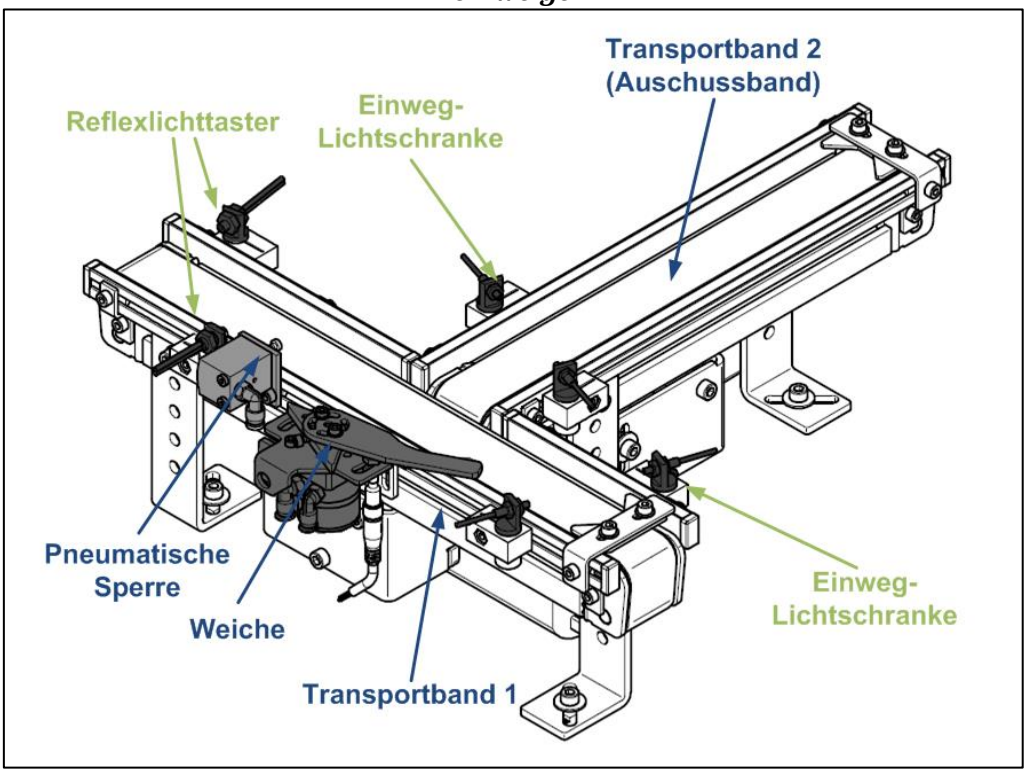
Das Modul dient dazu Werkstücke anhand ihrer Höhe zu klassifizieren. In einem bestimmten, hardwaretechnisch festgelegten Intervall (>25mm: rot und silbern mit Deckel) gibt der Sensor einen positiven Wert (true) zurück.

Weiche



Das Modul dient dazu Werkstücke umzuleiten oder vorbeizulassen. In der Ausgangsstellung können Werkstücke ungehindert passieren. Beide Endlagen der Weiche sind über Sensoren erfassbar.

Verzweigen



Das Modul dient dazu die Werkstücke über zwei Transportbänder entsprechend ihrer vorhergehenden Klassifikation weiterzuleiten. Zur Erkennung von Werkstücken sind mehrere Lichttaster und Lichtschranken installiert. Um das Klassifizieren der Werkstücke durch das Modul *Unterscheiden* zu vereinfachen ist zusätzlich eine pneumatische Sperre installiert.

Ablauf

Zu Anfang befindet sich kein Werkstück in der Station und sowohl Sperre als auch Weiche befinden sich in ihrer Ausgangsstellung (Sperre ausgeschoben, Weiche nicht ablenkend). Beide Bänder sind ausgeschaltet. Zudem wird kein Freigabesignal gegeben. Wenn der Ausgangszustand hergestellt ist, beginnt die Anlage mit folgender Routine:

Zunächst muss ein Freigabesignal an die vorherige Station gegeben werden, damit Werkstücke in die Station gelangen können. Wird ein Werkstück in der ersten Lichtschranke erkannt, wird das Freigabesignal sofort zurückgenommen. Darauf wird nach kurzer Wartezeit das Hauptband angeschaltet. Nachdem das Werkstück an der Sperre angekommen ist, wird das Band ausgeschaltet und das Werkstück wird in seiner Höhe vermessen. Rote und silberne Werkstücke mit Deckel sollen auf dem Hauptband verbleiben. Auf Basis dieser Information wird die Weiche entsprechend geschaltet und ggf. das Ausschussband angeschaltet. Ist die Weiche in der richtigen Stellung wird die Sperre eingezogen und das Hauptband wieder eingeschaltet.

Der Fall, dass eines oder beide Bänder voll werden, wird hier nicht weiter betrachtet.

Hinweise

- Alle Werkstücke ohne Deckel sind mit Öffnung nach unten einzulegen, damit die Höhenmessung richtig durchgeführt werden kann.
- Bei den Messungen sollte sich das Werkstück eine kurze Zeit (z. B. drei Sekunden) in Ruhe bei dem Sensor befinden, bevor der Wert abgefragt wird.
- Bei Vorgängen deren Endposition nicht durch Endlagensensoren überprüft werden können, müssen Sie zur Annahme des Erreichens der Endlage Zeitmessungen durchführen. Gehen Sie zunächst von ausreichend großen Werten aus (z. B. fünf bis zehn Sekunden), bevor Sie die Zeiten später optimieren.
- Auf silberne Werkstücke ohne Deckel wird auf Grund des (in diesem Falle) unzuverlässigen Höhensensors bei der Einzelabnahme dieser Station verzichtet.

Wichtige Variablen

Eingänge (Rückmeldung der Anlage)

Bit	Variablenname	Beschreibung
0	sensoren.werkstueck_am_anfang	Reflexlichttaster am Anfang des Bandes detektiert ein Werkstück
1	sensoren.werkstueck_an_sperre	Reflexlichttaster an der pneumatischen Sperre detektiert ein Werkstück
2	sensoren.hoehe_ok	Werkstückhöhe liegt in einem bestimmten Intervall (>25mm: rot und silbern mit Deckel)
3	sensoren.weiche_nicht_ablenkend	Weiche lässt Werkstücke passieren
4	sensoren.ausschussband_nicht_voll	Das Ausschussband ist nicht voll (Lichtschranke **nicht** durchbrochen)
5	sensoren.werkstueck_am_ende	Lichtschranke am Hauptbandende **nicht** durchbrochen

Ausgänge (Befehle von Benutzer an Anlage)

Bit	Variablenname	Beschreibung
0	aktoren.hauptband_an	Hauptband anschalten
1	aktoren.ausschussband_an	Ausschussband anschalten
2	aktoren.sperre_einziehen	Pneumatische Sperre einziehen (Werkstücke durchlassen)
3	aktoren.weiche_lenke_ab	Weiche schalten, so dass die Werkstücke auf das Ausschussband gelenkt werden
7	aktoren.station_frei	Freigabesignal an Vorgängerstation senden

Abnahme

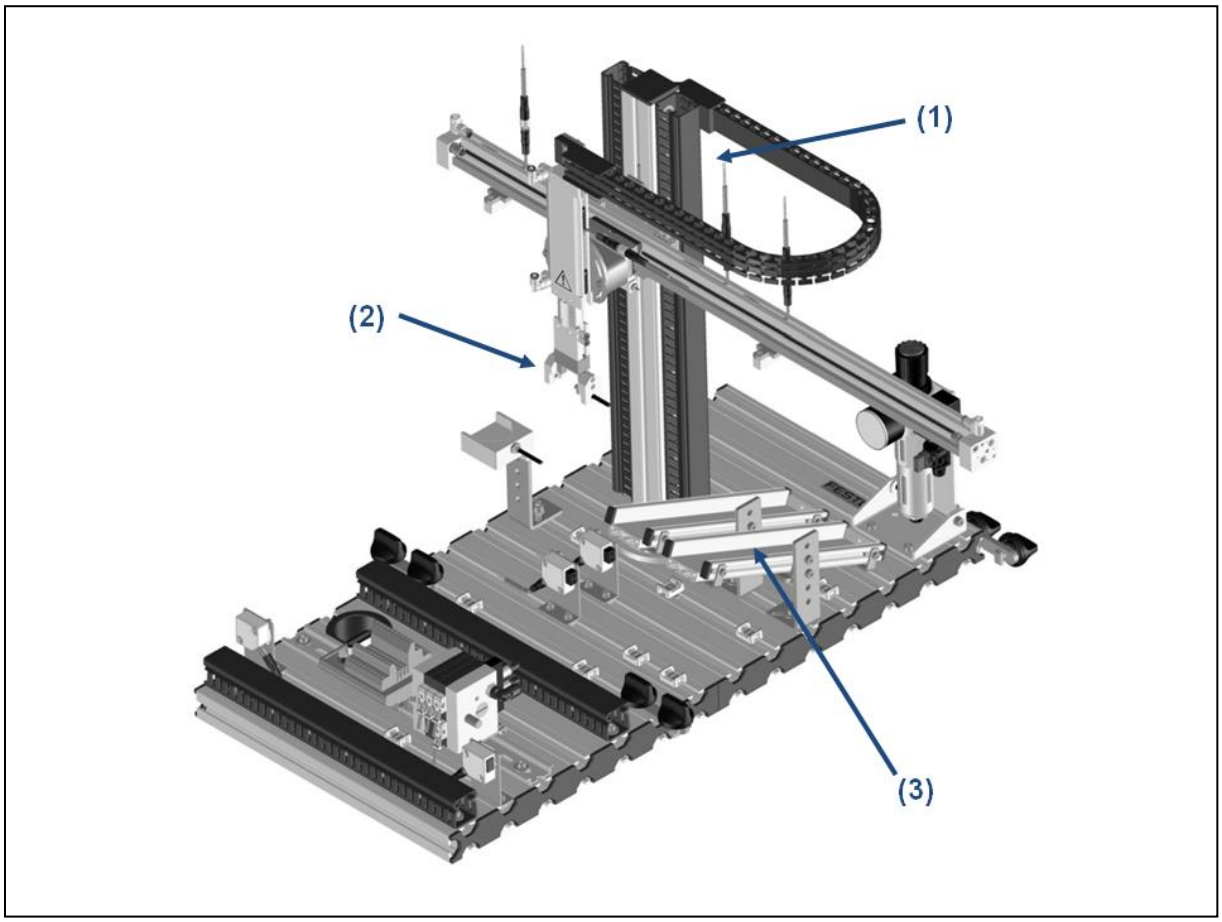
Zur Abnahme der Station sind folgende Eigenschaften zu erfüllen:

- Ausgangssituation laut Ablaufbeschreibung.
- Werkstücke werden so sortiert, dass alle roten und silbernen Werkstücke mit Deckel auf dem Hauptband weitertransportiert, alle anderen auf das Ausschussband befördert werden. Beachten Sie hierbei die unterschiedlichen Höhen der Werkstücke verschiedener Farben!
- Freigabesignal darf nur gegeben werden, wenn Abstellplatz am Bandanfang nicht belegt ist.

Es werden vier Werkstücke (schwarz ohne Deckel, rot ohne Deckel, rot mit Deckel, schwarz mit Deckel) in zufälliger Reihenfolge einzeln auf das Band gelegt. Neue Werkstücke werden erst auf das Band gelegt, wenn das Freigabesignal an die Vorgängerstation gegeben wurde. Auf silberne Werkstücke wird verzichtet, da der Höhensensor hier in manchen Fällen nicht korrekt arbeitet.

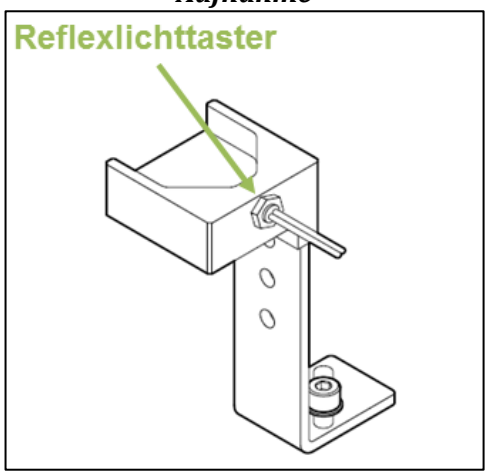
4. Station Handhaben

Herzstück dieser Station ist ein *Zwei-Achs-Kran (PicAlfa) (1)*, der mittels eines *pneumatischen Greifers (2)* Werkstücke von der Aufnahme abholen kann und sie anschließend auf eine der beiden *Rutschen (3)* oder der Folgestation ablegt. Im Greifer ist ein optischer Sensor angebracht, der zwischen „schwarzen“ und „nicht-schwarzen“ Werkstücken unterscheiden kann.



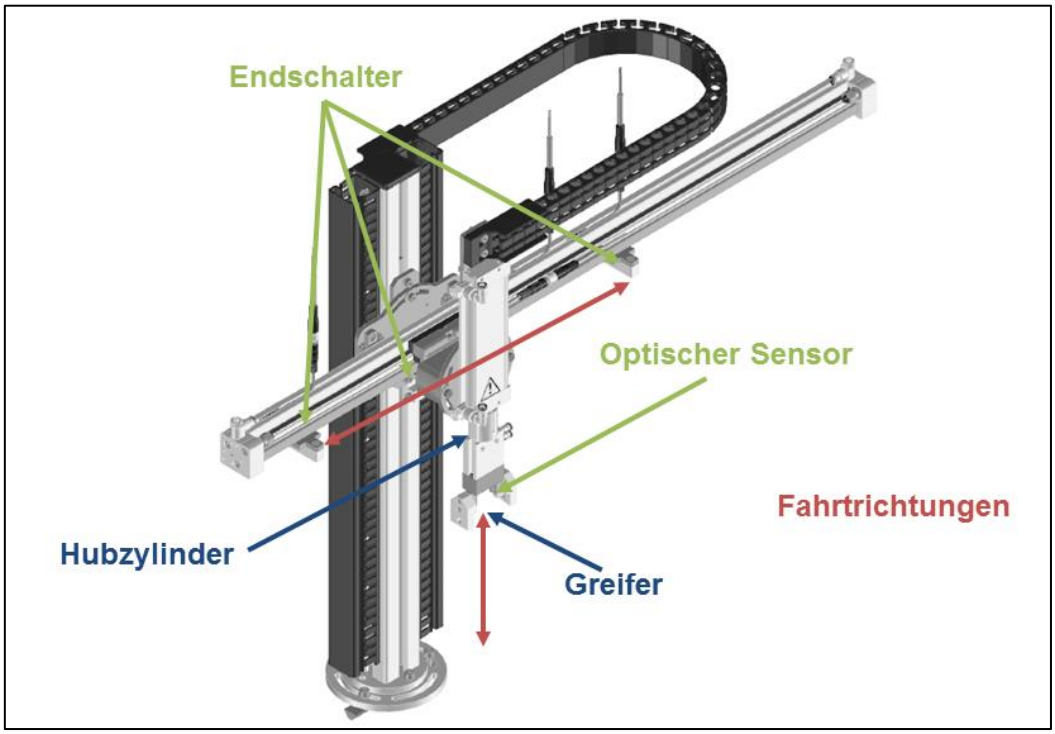
Module

Aufnahme



Das Modul dient dazu die Werkstücke in einer definierten Position zu halten, so dass sie vom Kran beziehungsweise dessen Greifer aufgenommen werden können. Die Anwesenheit eines Werkstücks kann über einen Lichttaster erkannt werden. Zur Kollisionsvermeidung mit der Vorgängerstation werden Werkstücke erst erkannt, wenn der Umsetzer der Vorgängerstation sich in der Ausgangsposition befindet.

PicAlfa



Das Modul besteht aus einem 2-Achs-Kran, der über einen Greifer verfügt, um Werkstücke aufnehmen zu können. Für drei Positionen des Krans existieren Sensoren. Zusätzlich ist im Greifer ein Sensor installiert, um schwarze und nicht-schwarze Werkstücke zu unterscheiden. Die vertikale Position des Greifers kann ebenfalls über Sensoren abgefragt werden.

Ablauf

Zu Anfang befindet sich in der Aufnahme kein Werkstück, der Greifer ist geöffnet in seiner oberen Endlage und der Kran steht bei der Folgestation Zudem wird kein Freigabesignal gegeben. Wenn der Ausgangszustand hergestellt ist, beginnt die Anlage mit folgender Routine:

Es wird ein Freigabesignal gegeben, um Werkstücke anzufordern. Sobald in der Aufnahme ein Werkstück erkannt wird, wird das Freigabesignal an die Vorgängerstation sofort zurückgenommen. Beachten Sie, dass Werkstücke erst erkannt werden, wenn auch der Umsetzer der Vorgängerstation in seiner

Ausgangslage (beim Magazin) ist. Der Kran verfährt nun zur Aufnahme, der Greifer wird abgesenkt, das Werkstück gegriffen und der Greifer wieder nach oben gefahren. Anschließend wird überprüft, ob das Werkstück schwarz oder nicht schwarz ist. Darauf fährt der Kran bei schwarzen Werkstücken, zur Rutsche und legt diese dort ab. Ansonsten fährt der Krahn zur Folgestation und legt das Werkstück dort ab (Freigabesignal der Folgestation nötig!). Danach fährt er wieder auf seine Ausgangsposition und gibt erneut das Freigabesignal an die Vorgängerstation.

Hinweise

- Wenn beide Pneumatikventile für den Kran angeschaltet sind, bewegt sich der Kran nicht.
- Für das Absenken und das Öffnen des Greifers werden Pneumatikzylinder eingesetzt. Das Heben und Schließen des Greifers geschieht mittels Rückstellfedern.
- Bei den Messungen sollte sich das Werkstück eine kurze Zeit (z. B. eine Sekunden) in Ruhe bei dem Sensor befinden, bevor der Wert abgefragt wird.
- Bei Vorgängen deren Endposition nicht durch Endlagensensoren überprüft werden können, müssen Sie zur Annahme des Erreichens der Endlage Zeitmessungen durchführen. Gehen Sie zunächst von ausreichend großen Werten aus (z. B. fünf Sekunden), bevor Sie die Zeiten später optimieren.
- Beachten Sie, dass das Freigabesignal der Folgestation hardwaregebunden standardmäßig gegeben wird (folgestation_belegt = 0), selbst wenn die Folgestation nicht läuft oder programmiert wird.

Wichtige Variablen

Eingänge (Rückmeldung der Anlage)

Bit	Variablenname	Beschreibung
0	sensoren.werkstueck_vorhanden	In der Werkstückaufnahme liegt ein Werkstück und die Vorgängerstation ist im Ausgangszustand (am Magazin)
1	sensoren.kran_bei_aufnahme	Der Kran ist bei der Aufnahme
2	sensoren.kran_bei_folgestation	Der Kran ist beim hinteren Sensor (Folgestation)
3	sensoren.kran_bei_rutsche	Der Kran ist beim vorderen Sensor (1. Rutsche)
4	sensoren.greifer_unten	Der Greifer ist unten
5	sensoren.greifer_oben	Der Greifer ist oben
6	sensoren.werkstueck_nicht_schwarz	Der Sensor im geschlossenen Greifer erkennt ein nicht schwarzes Werkstück
7	sensoren.folgestation_belegt	Die Folgestation ist nicht frei

Hinweis: Das Freigabesignal kann in dieser Station auch durch den Schlüsselschalter an der Bedienkonsole vorgetäuscht werden, falls die Folgestation beispielsweise nicht programmiert wird.

Ausgänge (Befehle von Benutzer an Anlage)

Bit	Variablenname	Beschreibung
0	aktoren.kran_zu_aufnahme	Kran in Richtung Aufnahme (Vorgängerstation) fahren
1	aktoren.kran_zu_folgestation	Kran in Richtung Folgestation fahren
2	aktoren.greifer_abwaerts	Greifer nach unten fahren
3	aktoren.greifer_oeffnen	Greifer öffnen
7	aktoren.station_frei	Vorgängerstation ein Freigabesignal geben

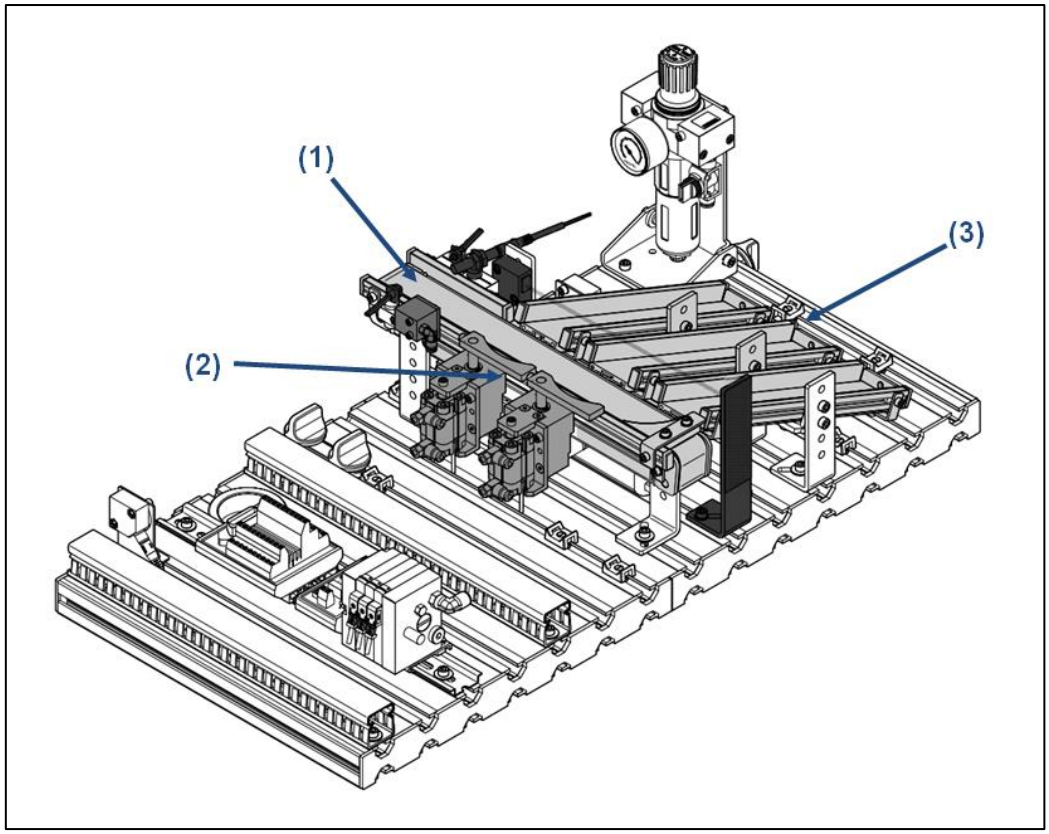
Abnahmen
Zur Abnahme der Station sind folgende Eigenschaften zu erfüllen:

- Ausgangssituation laut Ablaufbeschreibung.
- Freigabesignal für die Vorgängerstation wird zurückgenommen, sobald Werkstück auf Aufnahme liegt und Vorgängerstation in Ausgangsposition (am Magazin) ist.
- Kran fährt nicht in Richtung der Folgestation, wenn das Freigabesignal der Folgestation nicht gegeben wurde.
- Der Kran transportiert ein schwarzes Werkstück auf die Ausschussrutsche und alle anderen zur Folgestation (Versuch mit schwarzem Werkstück darf wiederholt werden, da Sensor nicht sehr zuverlässig ist).

Es werden vier Werkstücke (schwarz ohne Deckel, rot/silbern ohne Deckel, rot/silbern mit Deckel, schwarz mit Deckel) in zufälliger Reihenfolge einzeln in die Aufnahme gelegt. Neue Werkstücke werden erst auf die Aufnahme gelegt, wenn das Freigabesignal an die Vorgängerstation gegeben wurde.

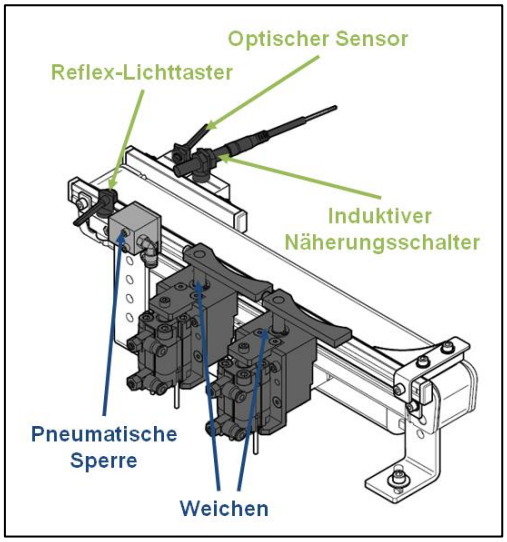
5. Station Sortieren

Beim Erreichen der Station „Sortieren“ wird das Werkstück zunächst am *Erkennungsmodul (1)* angehalten und eindeutig identifiziert. Anschließend wird es entsprechend der Vorgaben mithilfe der beiden *Weichen (2)* auf eine der drei *Rutschen (3)* einsortiert.



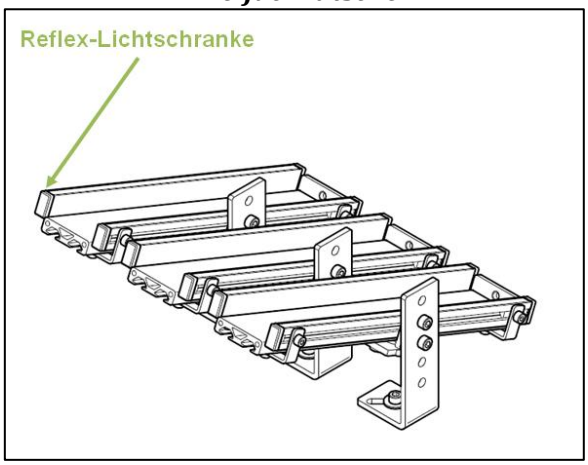
Module

Sortierband



Das Modul dient dazu Werkstücke zu klassifizieren und entsprechend weiterzuleiten. Zur Klassifikation von Werkstücken verfügt das Modul über einen induktiven Sensor, der metallische von nicht-metallischen Werkstücken unterscheiden kann; sowie einen optischen Sensor, der schwarze von nicht-schwarzen Werkstücken unterscheiden kann. Zur Klassifikation können die Werkstücke mittels einer pneumatischen Sperre angehalten werden. Das Einsortieren der Werkstücke in die passenden Rutschen geschieht durch zwei Weichen, deren Endlagen ebenfalls erfasst werden. Werkstücke am Anfang des Moduls werden zusätzlich über einen Lichttaster erkannt.

Dreifachrutsche



Das Modul besitzt drei Rutschen, in welche die Werkstücke einsortiert werden. Eine volle Rutsche wird über eine Lichtschranke erkannt.

Ablauf

Zu Anfang befindet sich kein Werkstück auf der Station, das Band ist aus und sowohl Sperre als auch Weichen in ihrer Ausgangsstellung (Sperre ausgeschoben, Weichen nicht ablenkend). Zudem wird kein Freigabesignal gegeben. Wenn der Ausgangszustand hergestellt ist, beginnt die Anlage mit folgender Routine:

Es wird der Vorgängerstation ein Freigabesignal weitergegeben, um ein Werkstück anzufordern. Sobald von der ersten Lichtschranke ein Teil erkannt wird, wird das Freigabesignal zurückgenommen und das Hauptband eingeschaltet. Nach kurzer Zeit ist das Werkstück an der Sperre angekommen. Dort wird an den Sensoren die Eigenschaft des Werkstückes überprüft. Währenddessen sollte das Hauptband ausgeschaltet sein, damit sich das Werkstück bei der Messung in Ruhe befindet. Im Folgenden werden die Weichen entsprechend der getroffenen Sortierentscheidung geschaltet. Sobald sich die Weichen in der richtigen Stellung befinden, wird die Sperre eingefahren, um das Werkstück passieren zu lassen. Nach einigen Sekunden wird die Station wieder in ihren Ausgangszustand versetzt.

Der Fall, dass die Rutschen voll werden, muss hier nicht weiter betrachtet werden.

Hinweise

- Bei Vorgängen deren Endposition nicht durch Endlagensensoren überprüft werden können, müssen Sie zur Annahme des Erreichens der Endlage Zeitmessungen durchführen. Gehen Sie zunächst von ausreichend großen Werten aus (z.B. fünf Sekunden), bevor Sie die Zeiten später optimieren.
- Die Lichtschranke überprüft den Arbeitsraum direkt vor dem ersten Stopper nicht.
- Die Werkstücke sollen entsprechend der folgenden Tabelle sortiert werden:

Werkstück	Rot	Schwarz	Metallisch
Rutsche	Erste (Weiche 1)	Mittlere (Weiche 2)	Letzte

Wichtige Variablen

Eingänge (Rückmeldung der Anlage)

Bit	Variablenname	Beschreibung
0	sensoren.werkstueck_am_anfang	Werkstück am Anfang des Bandes detektiert
1	sensoren.werkstueck_metallisch	Werkstück an Sperre ist metallisch
2	sensoren.werkstueck_nicht_schwarz	Werkstück an Sperre ist nicht schwarz
3	sensoren.rutschen_voll	Eine der drei Rutschen ist voll
4	sensoren.weiche1_nicht_ablenkend	Weiche 1 ist nicht geschaltet (lenkt nicht ab)
5	sensoren.weiche1_ablenkend	Weiche 1 ist geschaltet (lenkt ab)
6	sensoren.weiche2_nicht_ablenkend	Weiche 2 ist nicht geschaltet (lenkt nicht ab)
7	sensoren.weiche2_ablenkend	Weiche 2 ist geschaltet (lenkt ab)

Ausgänge (Befehle von Benutzer an Anlage)

Bit	Variablenname	Beschreibung
0	aktoren.band_an	Band anschalten
1	aktoren.weiche1_lenke_ab	Erste Weiche schalten, so dass die Werkstücke auf die erste Rutsche geleitet werden
2	aktoren.weiche2_lenke_ab	Zweite Weiche schalten, so dass die Werkstücke auf die zweite Rutsche geleitet werden
3	aktoren.sperre_einziehen	Pneumatische Sperre einziehen, also Werkstücke durchlassen
7	aktoren.station_frei	Freigabesignal an Vorgängerstation senden

Abnahme

Zur Abnahme der Station sind folgende Eigenschaften zu erfüllen:

- Ausgangszustand laut Ablaufbeschreibung.
- Band wird nach Sortiervorgang wieder ausgeschaltet.
- Station sortiert schwarze, rote und metallische Bauteile (mit oder ohne Deckel) in die drei unterschiedlichen Rutschen (welche Rutsche für welche Farbe verwendet wird, entspricht der Vorgabe).
- Das Freigabesignal wird an die Vorgängerstation gegeben, wenn die Station für die Aufnahme neuer Werkstücke bereit ist.

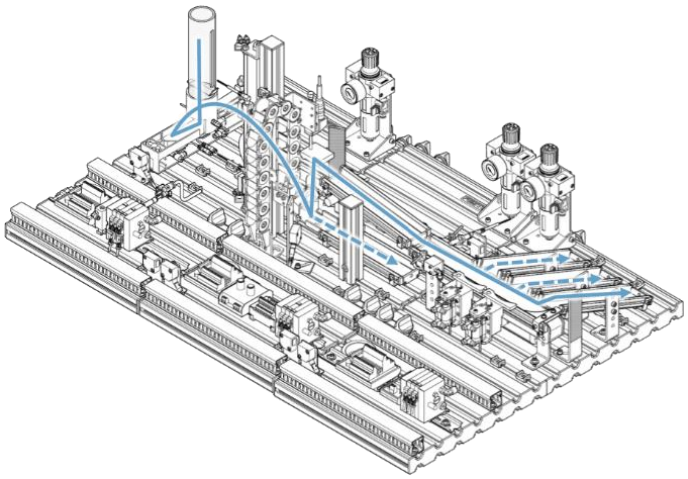
Es werden sechs Werkstücke (je zwei rote, schwarze und silberne) in zufälliger Reihenfolge einzeln auf das Band gelegt. Neue Werkstücke werden erst auf das Band gegeben, wenn das Freigabesignal an die Vorgängerstation gegeben wurde.

Anlagenbeschreibung

Im Anlagenpraktikum gibt es zwei verschiedene Anlagenkombinationen die je aus drei Stationen bestehen. Um die Teambonuspunkte zu erlangen, müssen je nach Anlagenkombination bestimmte Eigenschaften bei Durchführung eines festgelegten Testfalls erfüllt werden.

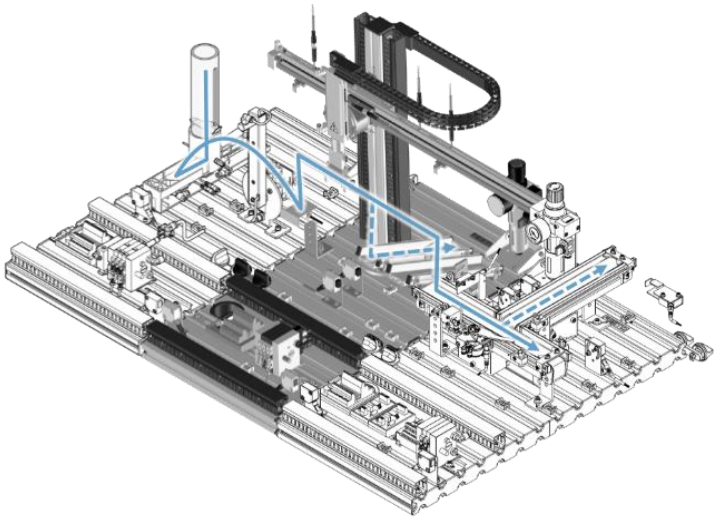
Kombination 1: Verteilen – Prüfen – Sortieren

Um den Teambonus bei dieser Anlage zu erlangen, müssen ***nach Abnahme der Einzelanlagen*** alle vier zufällig ausgewählte Werkstücke richtig sortiert werden. Unter den eingelegten Werkstücken muss sich mindestens ein Werkstück befinden, dass in die Sortierrutschen der letzten Station gelangen soll. In welche Sortierrutschen das Werkstück dabei gelangt, ist für die Auswahl unerheblich. Es dürfen keine Kollisionen zwischen den Stationen auftreten.

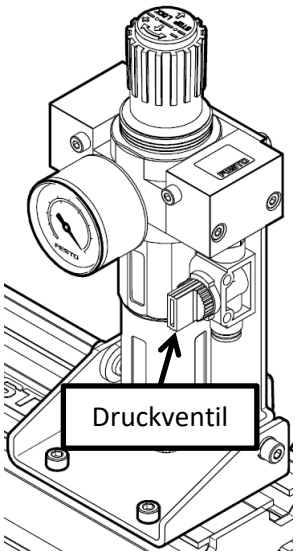


Kombination 2: Verteilen – Handhaben – Trennen

Um den Teambonus bei dieser Anlage zu erlangen, müssen ***nach Abnahme der Einzelanlagen*** alle vier eingelegten Werkstücke richtig sortiert werden. Es werden keine silbernen Werkstücke verwendet. Es dürfen keine Kollisionen zwischen den Stationen auftreten.



Sicherheitshinweise



- Sofern ein manueller Eingriff notwendig ist, schließen Sie auf jeden Fall das Druckventil und öffnen Sie dieses nur, wenn sich weder Werkstücke noch Körperteile im Gefahrenbereich befinden. Beachten Sie, dass trotz geschlossenem Ventil Aktoren kaum zu bewegen sein können.
- Versuchen Sie insbesondere den Aufzug der Station „Prüfen“ nicht gewaltsam zu bewegen!
- Halten Sie während des Betriebs der Anlage Ihre Hände von allen beweglichen Teilen fern.
- Verklemmen sich Teile der Anlage betätigen Sie die Stop-Taste auf der Bedienkonsole und setzen Sie das Programm zurück.
- Um Kollisionen mit anderen Stationen zu vermeiden, geben Sie das Freigabesignal per Schlüssel nur, wenn der Partner der Folgestation informiert ist oder an der Nachfolgestation nicht programmiert wird. Des Weiteren soll ein Freigabesignal nur gegeben werden, wenn die Anlage bereit ist ein Werkstück aufzunehmen.