

<b>1</b>	<b>Allgemeines zum Programmieren.....</b>	<b>4</b>
1.1	<i>Pseudocode.....</i>	4
1.2	<i>Flussdiagramm.....</i>	5
1.3	<i>Struktogramm.....</i>	6
1.3.1	Einzelaktion.....	6
1.3.2	Sequenz .....	6
1.3.3	Auswahlaktion (Alternative) .....	7
1.3.4	mehrfache Verzweigung .....	7
1.3.5	Iteration/Schleife .....	7
1.4	<i>Interaktives Programmieren .....</i>	8
<b>2</b>	<b>Die Programmiersprache C.....</b>	<b>9</b>
2.1	<i>Aufbau eines C-Programmes .....</i>	9
2.2	<i>Die Funktion main.....</i>	10
2.2.1	Blöcke.....	10
2.2.2	Anlegen von Variablen.....	10
2.2.3	Konstanten.....	10
2.2.4	Zuweisungen .....	11
<b>3</b>	<b>Datentypen in C.....</b>	<b>12</b>
<b>4</b>	<b>Ein- und Ausgabefunktionen in C .....</b>	<b>13</b>
4.1	<i>Ausgabefunktion printf() .....</i>	13
4.1.1	Minimale Feldbreite .....	13
4.1.2	Genauigkeitsangabe .....	14
4.1.3	Bündigkeit und Vorzeichenausgabe.....	14
4.2	<i>Eingabefunktion scanf() .....</i>	16
4.2.1	Tastaturpuffer und Besonderheiten von scanf() .....	17
4.2.2	getchar() und putchar().....	19
4.2.3	getch() und getche() .....	19
<b>5</b>	<b>Bedingte Anweisungen.....</b>	<b>20</b>
5.1	<i>if-else .....</i>	20
5.2	<i>Mehrfachauswahl: switch .....</i>	21
5.3	<i>Schleifen .....</i>	23
5.3.1	while-Schleife.....	23
5.3.2	for-Schleife.....	25
5.3.3	do-while.....	26
5.3.4	Geschachtelte Schleifen .....	26
5.3.5	break-Anweisung .....	27
5.3.6	continue-Anweisung.....	27
<b>6</b>	<b>Ausdrücke und Operatoren.....</b>	<b>28</b>
6.1	<i>Ausdrücke .....</i>	28
6.2	<i>Operatoren .....</i>	29
6.2.1	Arithmetische Operatoren .....	29
6.2.2	Vergleichsoperatoren .....	30
6.2.3	Logische Operatoren .....	30
6.2.4	Bitoperatoren.....	31
6.2.5	Zuweisungsoperatoren .....	32
6.2.6	Übrige Operatoren.....	32
6.2.7	Prioritäten von Operatoren.....	33
<b>7</b>	<b>Funktionen .....</b>	<b>34</b>
7.1	<i>Funktionsprototypen und -definitionen .....</i>	34
7.1.1	Gültigkeitsbereiche und Sichtbarkeit .....	35
7.1.2	Funktionsschnittstelle.....	37

7.1.3	Übergabe per Wert .....	37
7.1.4	Übergabe per Referenz .....	37
7.1.5	Übergabe per Zeiger .....	38
7.1.6	Rückgabewerte .....	39
7.1.7	Rekursiver Funktionsaufruf .....	39
7.2	Die Funktion <i>main()</i> .....	40
<b>8</b>	<b>Felder und Strings .....</b>	<b>41</b>
8.1	Felder .....	41
8.1.1	Eindimensionale Felder .....	41
8.1.2	Mehrdimensionale Felder .....	43
8.2	Zeichenketten ( <i>Strings</i> ) .....	43
8.2.1	Definition und Zuweisung .....	43
8.2.2	Ein- und Ausgabe .....	44
8.2.3	Initialisierung von Strings .....	44
8.2.4	Funktionen zur Stringmanipulation .....	45
<b>9</b>	<b>Zeiger .....</b>	<b>47</b>
9.1	Zeiger und Adressen .....	47
9.2	Zeiger und Funktionen .....	48
9.3	Zeiger und Arrays .....	49
9.4	Zeigervektoren .....	50
9.5	Argumentbehandlung .....	50
<b>10</b>	<b>Strukturen und Aufzählungstypen .....</b>	<b>53</b>
10.1	Strukturen .....	53
10.2	Zeiger und Strukturen .....	54
10.3	Aufzählungstypen .....	55
<b>11</b>	<b>Dynamische Speicherverwaltung .....</b>	<b>56</b>
<b>12</b>	<b>Dateibehandlung in C .....</b>	<b>58</b>
12.1	Allgemeines .....	58
12.2	Eine Datei öffnen .....	58
12.3	Schließen der Datei .....	59
12.4	Schreiben und Lesen .....	59
12.4.1	Formatierte Dateiausgabe .....	60
12.4.2	Formatierte Dateieingabe .....	60
12.4.3	Zeicheneingabe und -ausgabe .....	60
12.4.4	Zeicheneingabe .....	60
12.4.5	Zeichenausgabe .....	61
12.4.6	Direkte Dateieingabe und -ausgabe .....	63
12.4.7	Direktzugriff .....	64
<b>13</b>	<b>Prüfen von Programmen .....</b>	<b>66</b>
13.1	Arten von Fehlern .....	66
13.1.1	Syntaktische Fehler .....	66
13.1.2	Laufzeitfehler .....	66
13.1.3	Logische Fehler .....	66
13.2	Testen .....	66
13.2.1	Feststellen von Fehlern .....	67
13.2.2	Korrigieren der Fehler .....	67
<b>14</b>	<b>Such- und Sortialgorithmen .....</b>	<b>68</b>
14.1	Laufzeitkomplexität von Algorithmen .....	68
14.2	Suchalgorithmen .....	72
14.2.1	sequentielle Suche .....	73
14.2.2	binäre Suche .....	74

14.3	<i>Sortieralgorithmen</i> .....	75
14.3.1	Bubble-Sort .....	75
14.3.2	Selection-Sort .....	76
14.3.3	Insertion-Sort .....	77
14.3.4	Quick-Sort .....	78
14.3.5	Merge-Sort .....	79
14.3.6	Zusammenfassung .....	80
<b>15</b>	<b>Dynamische Datenstrukturen</b> .....	<b>81</b>
15.1	<i>Einfach verkettete Listen</i> .....	81
15.1.1	Anwendungsgebiete .....	81
15.1.2	Aufbau .....	81
15.1.3	insertElement() .....	82
15.1.4	appendElement() .....	83
15.1.5	deleteFirstElement() .....	84
15.1.6	ClearAllElements() .....	85
15.1.7	deleteElement() .....	85
15.1.8	dataOutput() .....	87
15.1.9	Beispiel .....	87
15.2	<i>Doppelt verkettete Liste</i> .....	88
15.2.1	Aufbau .....	88
15.2.2	insertElementBefore() .....	88
15.2.3	insertElementBegin() .....	89
15.2.4	insertElementAfter() .....	89
15.2.5	appendElement() .....	90
15.2.6	deleteElement() .....	90
15.2.7	dataoutput() und clearAllElements() .....	91
15.3	<i>Der binäre Baum</i> .....	92
15.3.1	Begriffsbestimmungen .....	92
15.3.2	Anwendungsgebiete .....	94
15.3.3	Beispiel .....	95
15.4	<i>Spezielle Datenstrukturen</i> .....	99
<b>16</b>	<b>Der Präprozessor</b> .....	<b>100</b>
16.1	<i>Textersatz</i> .....	100
16.2	<i>Bedingte Übersetzung</i> .....	101
<b>17</b>	<b>Programmbausteine im Quellcode</b> .....	<b>104</b>
17.1	<i>Übergabe der Steuerung an andere Hauptprogramme</i> .....	104
17.2	<i>Objekt-Dateien</i> .....	105
17.3	<i>Das Entwickeln und Warten von Programmen (make) unter LINUX (UNIX)</i> .....	106

# 1 Allgemeines zur Softwareentwicklung

Der wichtigste Vorgang beim Programmieren ist die Entwicklung und saubere, fehlerfreie Beschreibung des Algorithmus. Zur übersichtlichen Darstellung eines Algorithmus sind Pseudocode, Flussdiagramm und Struktogramm sehr geeignete Werkzeuge.

**Definition:** Ein Algorithmus ist eine Verarbeitungsvorschrift, die so präzise ist, dass sie von einem mechanisch oder elektronisch arbeitenden Gerät ausgeführt werden kann.

## 1.1 Pseudocode

Der Pseudocode ist eine im Wesentlichen auf normale Umgangssprache basierende Darstellung. Mit seiner Hilfe kann ein Algorithmus sehr einfach in fast beliebiger Weise formuliert werden.

### Beispiel: Berechnung eines Prüfungsergebnisses

#### **PSEUDOCODE** Prüfungsergebnis

```
p einlesen
„Sie haben die Prüfung“ ausgeben
WENN  $p < max / 2$ 
    DANN { „nicht“ ausgeben }
    SONST { }
„bestanden“ ausgeben
```

#### **Konstante:**


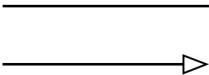
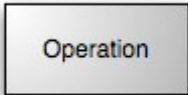
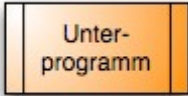
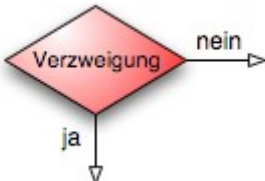
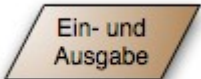
*max* = 100 maximal Punkteanzahl

#### **Variable:**

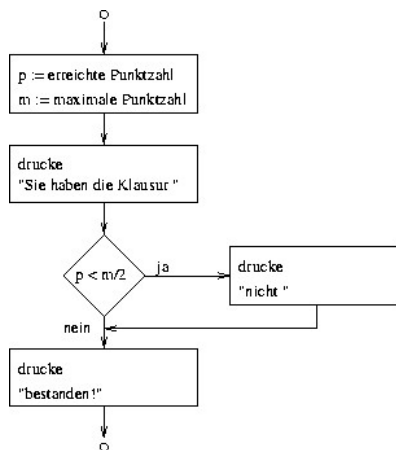
*p*: reell , erreichte Punkteanzahl

## 1.2 Flussdiagramm

Ein Flussdiagramm ist eine graphische Darstellung des Programmflusses. Dabei kommen die folgenden Symbole zum Einsatz:

	Start, Stopp, weitere Grenzpunkte
	Pfeil, Linie: Verbindung zum nächstfolgenden Element
	Rechteck: Operation
	Rechteck mit doppelten, vertikalen Linien: Unterprogramm aufrufen
	Raute: Verzweigung
	Parallelogramm: Ein- und Ausgabe

### Bsp: FLUSSDIAGRAMM Pruefungsergebnis



## 1.3 Struktogramm

Das Struktogramm (Nassi-Schneidermann-Diagramm) ist eine graphische Darstellungsform eines Algorithmus. Es besteht aus einem Namen, einem Deklarationsteil und einem Anweisungsteil.

Der **Deklarationsteil** ist eine Beschreibung der im Struktogramm verwendeten Konstanten und Variablen. Konstante Zahlenwerte sollten mit sinnvollen Namen belegt werden. Überall dort, wo einer dieser Zahlenwerte gebraucht wird, gibt man dann diesen Namen an. Dadurch werden zwei häufige Programmierfehler verhindert: Erstens sind Schreibfehler bei vielen Ziffern, wie sie besonders bei mehrmaligem Aufschreiben derselben Zahl oft auftreten, ausgeschlossen. Zweitens muss man bei einer späteren Änderung der Konstanten nur die Definition ändern. Es kann also nicht passieren, dass man das Ändern eines Zahlenwertes an einer der vielen Stellen des Programms, an denen er verwendet wird, vergißt.

Variablen wird nicht ein fixer Wert von Anfang an zugewiesen, sondern im Verlauf des Algorithmus werden sie mit Werten belegt, die jedoch wieder geändert werden können.

Der **Anweisungsteil**: Jede Aktion oder Anweisung wird in einem Struktogramm als Rechteck dargestellt. Die kleinsten Bausteine eines Algorithmus sind Einzelaktionen. Aufbauend auf diesen Einzelaktionen verwenden wir folgende Grundstrukturen zur Beschreibung von Algorithmen:

- Folge von Aktionen,
- Auswahl aus zwei Aktionen,
- Wiederholung von Aktionen.

### 1.3.1 Einzelaktion

Einfache Anweisungen (Einzelaktionen) werden in rechteckige Kästen gesetzt.

Anweisung
-----------

Eine Einzelaktion A kann sein:

- eine Wertzuweisung (zB: „Geburtsjahr = 1985“)
- eine Eingabe (zB: „Name einlesen“)
- eine Ausgabe (zB: „Name, Alter ausgeben“)

### 1.3.2 Sequenz

- Eine Sequenz ist die Aneinanderreihung von Teilalgorithmen bzw. Anweisungen
- Pro Anweisung gibt es ein Kästchen
- Die Anweisungen werden von oben nach unten abgearbeitet

Anweisung 1
-------------

Anweisung 2
-------------

:

Anweisung n
-------------

### 1.3.3 Auswahlaktion (Alternative)

- Sollen Anweisungen nur in Abhängigkeit von bestimmten Bedingungen ausgeführt werden, verwendet man eine Auswahl bzw. Verzweigung
- Ist die Bedingung wahr, so wird der „WAHR“-Block ausgeführt, ansonsten, der „FALSCH“-Block
- Jeder der Blöcke kann
  - eine Anweisung,
  - einen Teilalgorithmus, oder ein
  - ein ganzes Struktogramm beinhalten.

Bedingung	
wahr	falsch
Anweisung 1	Anweisung 2

### 1.3.4 mehrfache Verzweigung

Auch Fallunterscheidung genannt

- Mehrere Alternativen werden berücksichtigt
- Je nach dem, welchen Wert der Ausdruck enthält, wird einer der Fälle ausgewählt
- Trifft keiner der Fälle zu, wird der „SONST“-Block ausgeführt.

Fallausdruck					
Fall 1	Fall 2	Fall 3	Fall n-1	...	sonst
Anw. 1	Anw. 2	Anw. 3	...	Anw. n	—

### 1.3.5 Iteration/Schleife

Iteration ist die Wiederholung eines Teilalgorithmus in Abhängigkeit von einer Bedingung

- Iteration mit Eintrittsbedingung (while-Schleife)
  - Schleife mit Abfrage vor jedem Schleifendurchlauf
  - kopfgesteuert
- Wiederholung mit Zählvorschrift (for-Schleife)
  - Sonderfall der Iteration mit Eintrittsbedingung
  - Schleife mit vorgegebener Zählvorschrift bzw. Durchlaufzahl (Zählschleife)
- Iteration mit Wiedereintrittsbedingung.
  - Schleife mit Abfrage nach jedem Schleifendurchlauf
  - fußgesteuert

### 1.3.5.1 Iteration mit Eintrittsbedingung (while-Schleife)

- Am Anfang der Iteration steht eine Eintrittsbedingung
- Der anschließende Teilalgorithmus wird wiederholt, solange die Bedingung erfüllt ist
- Ist die Bedingung von Beginn an nicht erfüllt, kommt der Teilalgorithmus auch nicht zur Ausführung

Eintrittsbedingung	
	Anweisung

### 1.3.5.2 Iteration mit fixer Anzahl von Wiederholungen (for-Schleife)

- wird in Fällen verwendet, in denen feststeht, wie oft die Schleife durchlaufen werden soll
- die Laufbedingung wird am Beginn der Iteration überprüft
- die Schrittweite kann bei jeder Schleife unterschiedlich eingestellt werden

C :

<code>for (i=start ; i&lt;=ende ; i=i+1)</code>	
	Anweisung

Small Basic:

<code>For i = start To ende Step 1</code>	
	Anweisung

### 1.3.5.3 Iteration mit Wiedereintrittsbedingung (do-while-Schleife)

- Der Teilalgorithmus wird ausgeführt, solange die Wiedereintrittsbedingung (Laufbedingung) erfüllt ist
- Ist die Bedingung nichtmehr erfüllt, wird die Wiederholung beendet
- Der Teilalgorithmus wird mindestens einmal ausgeführt.

Anweisung
Wiedereintrittsbedingung

## 1.4 Interaktives Programmieren

Nachdem ein Programm gestartet wird, führt der Computer die programmierten Aktionen (Berechnungen) mit den vorgegebenen Eingabedaten selbständig durch und liefert danach die Ergebnisse. Soll der Benutzer während des Ablaufs zusätzlich eingreifen und verändern (interagieren) können, so ist dies im Programm zusätzlich zu implementieren und ein sogenanntes interaktives Programm zu entwerfen.

Dabei sind 2 Regeln zu beachten:

- vor jeder Eingabe soll dem Benutzer mitgeteilt werden, welche Eingabe von ihm verlangt wird
- der Benutzer soll die Möglichkeit besitzen, nach erfolgter Berechnung zu entscheiden, ob eine erneute Berechnung durchgeführt werden soll.



## 2 Die Programmiersprache C

C wurde 1971 als Grundlage für das Betriebssystem UNIX in den USA entwickelt (UNIX ist zu über 90% in C geschrieben). 1978 wurde von Brian Kernighan und Dennis Ritchie eine eindeutige Sprachdefinition entwickelt. C ist mittlerweile von ANSI und ISO standardisiert. C ist sehr eng an Assembler angelehnt, aber trotzdem Hardware-unabhängig. Das bedeutet, Sie können maschinennahe Programme sehr leicht (aber nicht ganz ohne Aufwand) auf ein anderes System portieren. Sie benötigen dazu lediglich einen anderen Compiler, und Inline-Assembler Anweisungen (Assembleranweisungen innerhalb eines C-Programmes) müssen der neuen Hardware (Prozessor) angepasst werden.

### Geschichte

1971	C wird entwickelt
1978	Kernighan und Ritchie definieren die Sprache.
1983	ANSI und ISO standardisieren C.
1992	Bjarne Stroustrup entwickelt die Nachfolgesprache C++.

### 2.1 Aufbau eines C-Programmes

C-Programme haben keinen fixen Aufbau. Es gibt zwar gewisse Regeln, aber sonst sind dem Programmierer alle Freiheiten überlassen. Der folgende Beispiel-Aufbau ist daher nicht zwingend und kann durchaus verändert werden.

```
#include <stdio.h>      /* Header-Include */
#include <stdlib.h>     /* Header-Include */
#define PI 3.14159     /* symbolische Konstante */

int zahl1;             /* globale Variablen */
char Zeichen1;

int main(void)         /* Definition des Hauptprogrammes */
{
    int zahl2;         /* lokale Variable */

    /* hier stehen die Befehle des Hauptprogrammes */

    return EXIT_SUCCESS;
}
```

#### *Kommentarbildung:*

Eine von den Zeichen `/*` und `*/` eingerahmte Zeichenkette. Sie hat für den Compiler keine Bedeutung. Sie kann (fast) überall im Programmtext stehen.

Die `#include <...>` Anweisung teilt dem Compiler mit, welche Header-Dateien er einbinden soll. In den Header-Dateien und den dazugehörigen Bibliotheken stehen Anweisungen und Datentypen. Dies sind am Beispiel der `stdio.h` die Befehle für die Ein- und Ausgabe.

Alle unter `#define` angeführten Konstanten werden vom Präprozessor, der vor dem Compiler seine Arbeit aufnimmt, durch den entsprechenden Wert ersetzt.

## 2.2 Die Funktion main

Ein C-Programm besteht aus einer oder mehreren Funktionen. Für die spezielle Funktion `main` wurde definiert, dass sie immer als erstes ausgeführt wird. Sobald `main` beendet ist, wird auch das Programm beendet.

```
int main(void)
{
    Anweisung1;
    return EXIT_SUCCESS;
}
```

Ohne `main` ist kein Programm komplett, es lässt sich auch nicht compilieren oder gar ausführen.

### 2.2.1 Blöcke

Programme sind in Abschnitte unterteilt, wobei jeder dieser Abschnitte ein eigenständiges Stück Code darstellt. Diese werden mit `{` und `}` gekennzeichnet. `{` bedeutet so viel wie "Block Anfang" und `}` bedeutet "Block Ende".

### 2.2.2 Anlegen von Variablen

Um eine Variable verwenden zu können, muss sie zuerst deklariert werden. Dies erfolgt mittels Angabe des Datentyps und des Namens der Variablen.

### 2.2.3 Konstanten

Konstanten können als Variable angesehen werden, deren Wert nicht verändert werden kann. Ein typisches Beispiel dafür ist die Zahl `PI` (rund 3,141592654).

## Namensbildung

Eine Folge von Buchstaben, Ziffern und dem Zeichen `_` (Unterstrich), die mit einem Buchstaben oder `_` beginnt. Es ist zwischen Groß- und Kleinbuchstaben zu unterscheiden. Namen dürfen keine C-Schlüsselworte (reservierte Worte) sein.

Schlüsselwort	Einsatzgebiet
<code>auto</code>	Variablendeklaration
<code>break</code>	Sprunganweisung
<code>case</code>	Mehrfachauswahl
<code>char</code>	Variablendeklaration
<code>const</code>	Konstantendeklaration
<code>continue</code>	Sprunganweisung
<code>default</code>	Mehrfachauswahl
<code>do</code>	Schleifen
<code>double</code>	Variablendeklaration
<code>else</code>	Verzweigung
<code>enum</code>	Typdefinition
<code>extern</code>	Variablendeklaration
<code>float</code>	Variablendeklaration

for	Schleifen
goto	Sprunganweisung
if	Verzweigung
int	Variablendeklaration
long	Variablendeklaration
register	Variablendeklaration
return	Funktionen
short	Variablendeklaration
signed	Variablendeklaration
sizeof	Operator
static	Variablendeklaration
struct	Typdefinition
switch	Mehrfachauswahl
typedef	Typdefinition
union	Typdefinition
unsigned	Variablendeklaration
void	Variablendeklaration
volatile	Variablendeklaration
while	Schleifen

### Beispiele:

```
int zahl1, zahl2;
char Zeichen='A';
const float pi=3.141592;

int main(void)
{
    float gleitZahl1;
    zahl1=23;
    zahl2=zahl1;
    /* weitere Anweisungen */

    return EXIT_SUCCESS;
}
```

### Erklärung:

In einer Zeile können auch mehrere Variablen gleichen Types vereinbart werden, wenn man einen Beistrich zwischen ihre Namen setzt. Variablen können in jedem "Block" vereinbart werden.

## 2.2.4 Zuweisungen

Sobald eine Variable vereinbart ist, kann man ihr Werte zuweisen. Dazu schreibt man zuerst den Variablennamen, ein Gleichheitszeichen "=" und anschließend den zuzuweisenden Ausdruck.

Die Konstante pi trägt nun den Wert 3.141592; dieser kann zur Laufzeit des Programmes nicht verändert werden. Die Anweisung "pi=123;" würde zu einem Compiler-Fehler führen. Wenn Sie den Wert ändern möchten, so müssen Sie den Wert bei der Konstantendefinition verändern.

### 3 Datentypen in C

#### char

Er belegt 1 Byte und kann sowohl zur Darstellung von Zeichen (Buchstaben, Ziffern usw.) als auch zur Darstellung von ganzen Zahlen verwendet werden.

#### short, int und long

Alle 3 Typen dienen zur Darstellung von vorzeichenbehafteten ganzen Zahlen mit unterschiedlich großem Wertebereich. Die Wortbreite ist jedoch in C nicht festgelegt. Sicher sei nur, dass der Datentyp **long** nicht "kleiner" als **int** und **int** nicht "kleiner" als **short** ist.

#### float, double und long double

Diese 3 Typen dienen zur Darstellung von Gleitkommazahlen mit unterschiedlich großem Wertebereich und unterschiedlich grosser Genauigkeit. Auch hier legt C die Wortbreite nicht fest; es gilt lediglich die folgende Rangfolge bzgl. der Wortbreite:

**float** <= **double** <= **long double**

C-Typ	Bytes	Wertebereich (kleinster..größter Wert), Bemerkungen
void	.	leerer Typ, z.B. für Funktion ohne Ergebniswert
char	1	-128..+127
unsigned char	1	0..255
short	2	-32768..32767
unsigned short	2	0..65535
long	4	-2147483648..2147483647
unsigned long	4	0..4294967295
int, signed	*	maschinenabhängig
unsigned int	*	maschinenabhängig
float	4	3,4E-38.. 3,4E+38
double	8	1,7E-308.. 1,7E+308
long double	10	3,4 E-4932.. 1,1E+4932

## 4 Ein- und Ausgabefunktionen in C

### 4.1 Ausgabefunktion printf()

Die Funktion `printf` gibt die formatierten Informationen auf dem Standardausgabegerät, normalerweise dem Bildschirm, aus. Bei Verwendung von `printf` müssen Sie die Header-Datei für die Standardein-/ausgabe, `stdio.h`, einbinden.

```
#include <stdio.h>
printf( Formatstring[,Argumente,...]);
```

Der Formatstring kann auch sogenannte Formatanweisungen beinhalten, die mit einem `%`-Zeichen beginnen.

Spezifizierer	Bedeutung	konvertierte Typen
<code>%c</code>	Einfaches Zeichen	char
<code>%d</code>	Vorzeichenbehaftete Dezimalzahl	int, short
<code>%ld</code>	Große vorzeichenbehaftete Dezimalzahl	long
<code>%f, %e</code>	Fließkommazahl	float, double
<code>%s</code>	Zeichenstring	char arrays
<code>%u</code>	Vorzeichenlose Dezimalzahl	unsigned int, unsigned short
<code>%lu</code>	Große vorzeichenlose Dezimalzahl	unsigned long
<code>%x %X</code>	Hexadezimale Zahl	unsigned int
<code>%o</code>	Oktale Zahl	unsigned int
<code>%%</code>	Ausgabe des Prozentzeichens	

#### Zahlenbasis

Fügt man bei den Formaten `%o`, `%x` und `%X` das Zeichen `#` zwischen `%` und Kennbuchstaben ein, so erscheinen die betreffenden Werte bei der Ausgabe mit einer vorangestellten Kennung der Zahlenbasis.

Bsp.:

```
printf("Mit Kennung %#o %#x %#X",123,123,123); /* Mit Kennung 0173 0x7b 0x7B */
```

#### 4.1.1 Minimale Feldbreite

Die minimale Feldbreite legt die Mindestanzahl der auszugebenden Stellen fest. Die minimale Feldbreite wird, falls gewünscht, direkt hinter dem Prozentzeichen angegeben.

```
printf("%d\n%ld\n%4d\n", 17, 17, 17);
printf("%f\n%4f\n%16f\n%e\n", 3.141593, 3.141593, 3.141593,3.141593);
printf("%s\n%4s\n%12s\n", "string", "string", "string");
```

```

Ausgabe:
17
17
    17
3.141593
3.141593
        3.141593
3.141593e+000
string
string
        string

```

Ist die minimale Feldbreite größer als die Anzahl der auszugebenden Zeichen, werden zu Beginn der Ausgabe Leerzeichen eingefügt. Ist die minimale Feldbreite kleiner als die Anzahl der auszugebenden Zeichen, werden dennoch alle Zeichen ausgegeben. Bei der Ausgabe von Gleitkommazahlen zählt der Dezimalpunkt für die Anzahl der ausgegebenen Zeichen mit. Soll anstelle von führenden Leerzeichen führende Nullen ausgegeben werden, lässt sich das erreichen, indem der minimalen Feldbreite eine 0 vorangestellt wird.

```

printf("%06d", 100);
führt zu der Ausgabe:
000100

```

#### 4.1.2 Genauigkeitsangabe

Die Genauigkeitsangabe wird hauptsächlich bei der Ausgabe von Gleitkommazahlen verwendet und bestimmt die Anzahl der auszugebenden Nachkommastellen. Die Genauigkeitsangabe wird durch einen Punkt getrennt hinter der minimalen Feldbreite angegeben.

```

printf("%5.3f\n%11.6f\n\n", 3.1415, 3.1415);
führt z.B. zu der Ausgabe:
3.142
        3.141500

```

Wird keine Genauigkeit angegeben, werden defaultmäßig 6 Nachkommastellen ausgegeben. Wird als Genauigkeit explizit 0 angegeben, werden keine Nachkommastellen ausgegeben.

#### 4.1.3 Bündigkeit und Vorzeichenausgabe

Standardmäßig erfolgen Ausgaben rechtsbündig. Zum Beispiel führt

```

printf("%06d", 314);
zu der Ausgabe:
000314

```

Soll die Ausgabe linksbündig erfolgen, setzt man vor Angabe der minimalen Feldbreite ein Minuszeichen.

```

printf("%-6d", 314);
führt zu der Ausgabe:
314

```

Bei negativen Zahlen wird das Vorzeichen immer mitausgegeben. Soll das Vorzeichen auch bei positiven Zahlen ausgegeben werden, ist vor Angabe der minimalen Feldbreite ein Pluszeichen zu setzen (Erinnerung: ein Minuszeichen vor der minimalen Feldbreite führt zu einer linksbündigen Ausgabe).

```
printf("%06d\n", -314);  
printf("%+06d\n", 314);  
printf("%-06d\n", -314);  
führt zu der Ausgabe:  
-00314  
+00314  
-314
```

Es gibt auch die Möglichkeit, für Breite und Stellenzahl Variable anzugeben:

```
printf("%*.*f", feldbreite, genauigkeit, zahl);
```

Escape-Sequenzen bieten besondere Möglichkeiten zur Formatierung. Eine Escape-Sequenz besteht aus einem Backslash gefolgt von einem einfachen Zeichen.

Sequenz	Bedeutung
\a	alert (Akustisches Signal)
\b	Backspace
\n	NL (new line) Cursor springt zum Anfang der nächsten Zeile
\t	Horizontaler Tabulator
\\	Backslash
\?	Fragezeichen
\'	' wird ausgegeben
\"	" wird ausgegeben
\r	CR (carriage return) Cursor springt zum Anfang der aktuellen Zeile
\f	FF (formfeed) Seitenvorschub beim Drucker

Bsp:

```
#include <stdio.h>  
  
int main(void)  
{  
    int Zahl1=12;  
    char Zeichen1='A';  
  
    printf("Das ist Text, und er wird als solcher ausgegeben. \n");  
    printf("Der Wert der Variablen Zahl1 ist: %d \n",Zahl1);  
    printf("Der Wert der Variablen Zeichen1 ist: %c \n",Zeichen1);  
    printf("Der Wert der Variablen Zeichen1 ist: %d \n",Zeichen1);  
  
    return EXIT_SUCCESS;  
}
```

Ausgabe:

## 4.2 Eingabefunktion scanf()

Die Funktion `scanf()` ist eine C-Standardfunktion, mittels der Werte für Variablen von der Tastatur eingelesen werden können. Formal werden `scanf()` ein Formatstring (Zeichenkettenkonstante) und optional weitere Argumente übergeben.

```
scanf("Formatstring", [Argument_1, Argument_2, . . ., Argument_n])
```

Im Unterschied zu `printf()` sind die an `scanf()` übergebenen Argumente Speicheradressen entsprechend den Adressen der Variablen, in die Werte eingelesen werden sollen. Die Adresse einer Variablen `x` erhält man, indem man dem Variablennamen das 'kaufmännische Und' voranstellt, also `&x`.

Der Formatstring kann wie im Falle von `printf()` normalen Text und Formatanweisungen enthalten. Normaler Text im Formatstring muss auch von der Tastatur entsprechend eingelesen werden.

Beispiel 1:

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

Das Programm hält hier an und wartet auf eine Benutzereingabe. Die Eingabe muss vom Benutzer durch Drücken der *<Enter>*-Taste quittiert werden. Gibt der Benutzer zum Beispiel *5<Enter>* ein, hat die Variable `x` (Ganzzahl-Variable vorausgesetzt) anschließend den Wert 5.

Beispiel 2:

```
scanf("%d%d%f", &x, &y, &z);
```

Das Programm hält hier an und wartet auf die Eingabe dreier Zahlen (durch ein oder mehrere Leerzeichen getrennt) und anschließender Quittierung mittels *<Enter>*. Die ersten beiden Zahlen werden als Ganzzahlen eingelesen (`%d`), die dritte Zahl wird als Gleitkommazahl eingelesen (`%f`). Gibt der Benutzer zum Beispiel *5 7 1.32<Enter>* ein, hat `x` danach den Wert 5, `y` den Wert 7 und `z` den Wert 1.32.

Beispiel 3:

```
scanf("%d.%d.%d", &tag, &monat, &jahr);
```

Das Programm hält hier an und wartet auf die Eingabe einer Ganzzahl, danach auf die Eingabe eines Punktes (normaler Text innerhalb des Formatstrings), danach wiederum auf die Eingabe einer Ganzzahl, danach wieder auf die Eingabe eines Punktes u.s.w. Der Benutzer ist gezwungen, soll die Eingabe korrekt durchgeführt werden, die Punkte auch wirklich miteinzugeben, z.B. *13.8.1999<Enter>*. In diesem Fall sollte der Benutzer mittels `printf()`-Anweisung vor der Eingabe über das Eingabeformat informiert werden, z.B.:

```
printf("Bitte geben Sie ein Datum in der Form Tag.Monat.Jahr ein, z.B.  
11.2.1914\n");  
scanf("%d.%d.%d", &tag, &monat, &jahr);
```



## Formatanweisungen bei scanf

Formatangabe	Datenobjekt
%d, %i	ganze Dezimalzahl
%u	ganze Dezimalzahl ohne Vorzeichen
%o	ganze oktale Zahl
%x, %X	ganze Hex-Zahl
%c	ASCII-Zeichen
%f	Gleitkommazahl
%e, %E	Gleitkommazahl in Exp.schreibweise
%s	Zeichenkette

Als Formatanweisung zur Eingabe einer Ganzzahl kann entweder %d oder %i verwendet werden. Geläufiger ist %d. Soll eine ganze Zahl ohne Vorzeichen eingegeben werden, wird %u benutzt. Neben der dezimalen Eingabe können ganze Zahlen auch in oktaler (%o) oder hexadezimaler Darstellung (%x oder %X) eingegeben werden.

Ein Zeichen wird mit %c eingegeben.

Gleitkommazahlen werden mit %f (normale Dezimalpunktschreibweise) oder mit %e oder %E (Exponentialschreibweise) eingegeben.

Soll eine Zeichenkette eingegeben werden, lautet die entsprechende Formatanweisung %s.

Sind ganze Zahlen als **short** definiert, muss zwischen dem Prozentzeichen und dem Buchstaben d, i, u, o, x oder X der Buchstabe **h** eingefügt werden, z.B. %hd. Sind ganze Zahlen als **long** definiert, muss zwischen dem Prozentzeichen und dem Buchstaben d, i, u, o, x oder X der Buchstabe **l** eingefügt werden, z.B. %ld.

Sind Gleitkommazahlen anstatt als float als **double** definiert, sollte zwischen dem Prozentzeichen und dem Buchstaben f (bzw. e oder E) der Kleinbuchstabe **l** eingefügt werden, z.B. %lf. Sind Gleitkommazahlen anstatt als float als **long double** definiert, sollte zwischen dem Prozentzeichen und dem Buchstaben f (bzw. e oder E) der Großbuchstabe **L** eingefügt werden, z.B. %Lf.

### 4.2.1 Tastaturpuffer und Besonderheiten von scanf()

Solange die Eingabe nicht mittels <Enter> quittiert ist, kann sie überarbeitet bzw. geändert werden. Die einzelnen von der Tastatur eingegebenen Zeichen (bzw. Ziffern) werden in einen Tastaturpuffer geschrieben, der korrigierbar ist. Mit Quittierung durch die <Enter>-Taste werden die Zeichen an das Programm übermittelt. Da auch die <Enter>-Taste einem Zeichen entspricht, wird auch dieses Zeichen in den Tastaturpuffer geschrieben.

Die Eingabe von Werten von der Tastatur geschieht nach folgenden Regeln:

scanf liest bis zum nächsten nicht passenden Zeichen

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

liest zum Beispiel im Falle einer Benutzereingabe von *12a6<Enter>* wegen der Formatanweisung `%d` nur die 1 und die 2 aus dem Tastaturpuffer. Der Rest (*a* und *<Enter>*) verbleibt im Tastaturpuffer. Die Variable *x* hat nach dem Einlesen den Wert 12.

`scanf` liest bis zum nächsten Leerzeichen (Blank, Tab, Zeilentrenner)

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

liest zum Beispiel im Falle einer Benutzereingabe von *32 6<Enter>* nur die 3 und die 2 aus dem Tastaturpuffer. Der Rest (*Leerzeichen*, *6* und *<Enter>*) verbleibt im Tastaturpuffer. Die Variable *x* hat nach dem Einlesen den Wert 32.

`scanf` überliest führende Leerzeichen

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

liest zum Beispiel im Falle einer Benutzereingabe von *<Leerzeichen>84<Enter>* das führende Leerzeichen, die 8 und die 4 aus dem Tastaturpuffer, wobei das führende Leerzeichen verworfen wird und nur die 8 und die 4 an das Programm weitergegeben werden. Der Rest (*<Enter>*) verbleibt im Tastaturpuffer. Die Variable *x* hat nach dem Einlesen den Wert 84.

Auch ein vom letzten Lesen im Tastaturpuffer verbliebendes *<Enter>*-Zeichen gilt für den nächsten Lesevorgang als führendes Leerzeichen und wird damit aus dem Tastaturpuffer entfernt.

## ACHTUNG:

Die Regel vom Überlesen von führenden Leerzeichen gilt nicht im Zusammenhang mit dem Einlesen von Zeichen mittels `%c`. In diesem Fall gilt z.B. ein noch vorhandenes *<Enter>*-Zeichen im Tastaturpuffer als reguläres Zeichen (auch *<Enter>* hat einen Zeichencode). Dies führt in der Praxis oft zu einem logischen Fehler im Programmlauf. Angenommen, es soll mittels `scanf()` ein Zeichen in eine Variable *z* vom Typ `char` eingelesen werden.

Die entsprechende Programmanweisung wäre:

```
scanf("%c", &z);
```

Ist im Programm vorher schon einmal etwas eingelesen worden, hat dieser frühere Einlesevorgang dazu geführt, dass sich ein *<Enter>* im Tastaturpuffer befindet. Nun bedient sich `scanf("%c", &z);` dieses *<Enter>*-Zeichens, d.h. das Programm hält nicht, wie eigentlich erwartet, an, damit der Benutzer ein Zeichen eingeben kann, sondern läuft einfach weiter. Der Wert der Variablen *z* entspricht dem Zeichencode für das *<Enter>*-Zeichen.

Beispiel:

```
scanf("%d %c", &ia, &ic);
```

Eingabe: <i>70A&lt;Enter&gt;</i>	→	<i>ia</i> = 70, <i>ic</i> = A
Eingabe: <i>A70&lt;Enter&gt;</i>	→	<i>ia</i> = ?, <i>ic</i> = ?
Eingabe: <i>70 A&lt;Enter&gt;</i>	→	<i>ia</i> = 70, <i>ic</i> = A
Eingabe: <i>7 0A&lt;Enter&gt;</i>	→	<i>ia</i> = 7, <i>ic</i> = 0

### 4.2.2 getchar() und putchar()

Die Funktionen `getchar()` und `putchar()` sind C-Standardfunktionen zur Eingabe bzw. Ausgabe von Zeichen. Sie können alternativ zu `scanf()` bzw. `printf()` mit der Formatanweisung `%c` benutzt werden.

```
scanf("%c", &z); entspricht z = getchar();
printf("%c", z); entspricht putchar(z);
```

Beispiel:

```
/* Umwandeln eines Klein- in einen Großbuchstaben */
#include <stdio.h>

int main(void)
{
    char ch;

    printf("Geben Sie einen Kleinbuchstaben ein. ");
    ch = getchar();
    ch = ch - 32; /* Umwandeln in Großbuchstaben */
    printf("\nGroßbuchstabe = %c", ch);

    return EXIT_SUCCESS;
}
```

### 4.2.3 getch() und getche()

Die Funktionen `getch()` und `getche()` dienen der Eingabe von Zeichen über die Tastatur. Die Funktion `getche()` liest ein einzelnes Zeichen und zeigt das gelesene Zeichen anschließend auf dem Bildschirm an (Echo). Die Funktion `getch()` liest ein Zeichen, ohne es anzuzeigen.

Bekanntgemacht werden die beiden Funktionen dem Programm über die Headerdatei `conio.h`, die man mittels `"#include <conio.h>"` in das Programm einbindet.

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

int main(void)
{
    char c;
    printf("Geben Sie einen Buchstaben ein. ");
    c = getch();
    printf("\nSie haben folgendes Zeichen eingegeben: %c", c);

    return 0;
}
```

### ACHTUNG:

Bei einer Eingabe mit `getchar()` wird der Code der Eingabetaste in `"\n"` umgesetzt, während `getch()` oder `getche()` den Code `"\r"` zurückgibt.

# 5 Bedingte Anweisungen

## 5.1 if-else

Eine if-else-Anweisung wertet einen Ausdruck aus und verzweigt, je nach Wert des Ausdrucks, zu unterschiedlichen Programmteilen. Damit lassen sich Fallunterscheidungen programmieren.

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

Beispiel:

```
if (x > y)
    minimum = y;
else
    minimum = x;
```

Eine if-else-Anweisung wird ausgewertet, indem überprüft wird, ob der Ausdruck wahr (entspricht ungleich 0) ist. Falls dies der Fall ist, wird die Anweisung im if-Zweig ausgeführt. Sollen mehrere Anweisungen im if-Zweig ausgeführt werden, müssen diese innerhalb einer öffnenden und schließenden geschweiften Klammer als Anweisungsblock zusammengefasst werden.

Falls der Ausdruck 0 ist, wird der eventuell vorhandene else-Zweig ausgeführt. Fehlt der else-Zweig, wird das Programm direkt hinter der if-Anweisung fortgeführt.

```
if (x > y)
    x = 5;
y = 7;
```

In diesem Fall wird x auf den Wert 5 gesetzt, wenn vorher x größer als y war. y wird in jedem Fall auf den Wert 7 gesetzt.

if-else-Anweisungen können auch geschachtelt werden.

Beispiel:

```
if (x > 0)
    y = 1;
else
    if (x < 0)
        y = -1;
    else
        y = 100;
```

Falls x größer ist als 0, wird y auf 1 gesetzt, andernfalls wird, falls x kleiner ist als 0, wird y auf -1 gesetzt, andernfalls wird y auf 100 gesetzt. Das letzte else bezieht sich auf das vorhergehende if.

Durch eine einfache if-else-Anweisung ist es lediglich möglich, abhängig von einer Bedingung in maximal zwei Programmteile zu verzweigen. Geschachtelte if-else-Anweisungen gestatten eine größere Auswahl von Verzweigungsmöglichkeiten.

```

if (x == 1)
{
    /* Block 1 von Anweisungen */
}
else if (x == 2)
{
    /* Block 2 von Anweisungen */
}
else if (x == 3)
{
    /* Block 3 von Anweisungen */
}
else
{
    /* Block 4 von Anweisungen */
}

```

#### ACHTUNG:

Ein häufig vorkommender Fehler ist, bei Überprüfung auf Gleichheit anstelle des Vergleichsoperators == den Zuweisungsoperator = zu verwenden, z.B. statt `if (x == 5)` versehentlich zu schreiben: `if (x = 5)`. Der Ausdruck `x = 5` ist ungleich Null und damit immer logisch wahr (nicht das, was man eigentlich bezwecken wollte).

## 5.2 Mehrfachauswahl: switch

Neben verschachtelten if-else-Anweisungen gibt es zur Mehrfachauswahl die manchmal etwas übersichtlichere switch-Anweisung:

```

switch (x)
{
    case 1:
        /* Block 1 von Anweisungen */
        break;
    case 2:
        /* Block 2 von Anweisungen */
        break;
    case 3:
        /* Block 3 von Anweisungen */
        break;
    default:
        /* Block 4 von Anweisungen */
        break;
}

```

Der Typ des Argumentes x darf nur ein integraler Typ sein. Abhängig von seinem Wert werden die Anweisungen zwischen dem entsprechenden case und dem nächsten break ausgeführt. Da hier ein Anweisungsblock schon durch case und break definiert ist, müssen die Anweisungen zwischen case und break nicht mehr in geschweiften Klammern zusammengefaßt werden.

Im obigen Beispiel würde also der Anweisungsblock 1 zur Ausführung kommen, falls x den Wert 1 hat, der Block 2 würde zur Ausführung kommen, falls x den Wert 2 hat u.s.w. Sollte x keinen Wert zwischen 1 und 3 haben, würden die Anweisungen zwischen default und dem letzten break ausgeführt werden. Der default-Zweig ist optional.

#### ACHTUNG:

Da alle Anweisungen hinter der ersten übereinstimmenden case-Konstanten und dem nächsten break ausgeführt werden, ist im allgemeinen darauf zu achten, dass auch wirklich jedes case mit einem break abgeschlossen wird. Würde z.B. im obigen Beispiel das break für case 1 fehlen, würden für den Fall, dass x gleich 1 ist, nicht nur die Anweisungen aus Block 1, sondern zusätzlich auch noch die Anweisungen aus Block 2 ausgeführt werden. den Wert 'b' hat.

Häufig wird eine switch-Anweisung verwendet, um abhängig von Benutzereingaben unterschiedliche Aktionen auszuführen:

```
printf("Bitte geben Sie die auszuführende Aktion an.");
printf("\nZum Einfügen geben Sie bitte e ein");
printf("\nZum Ändern geben Sie bitte a ein");
printf("\nZum Löschen geben Sie bitte d ein");
printf("\nZum Sortieren geben Sie bitte s ein");
printf("\nIhre Eingabe bitte: ");

scanf("%c", &ch);

switch (ch)
{
    case 'e':
        einfuegen();
        break;
    case 'a':
        aendern();
        break;
    case 'd':
        loeschen();
        break;
    case 's':
        sortieren();
        break;
}
```

Dieser Programmteil könnte ein Auszug aus einem Personalverwaltungsprogramm sein, wo je nach Benutzereingabe Funktionen zum Einfügen, Ändern, Löschen oder Sortieren von Personaldaten aufgerufen werden.

## 5.3 Schleifen

### 5.3.1 while-Schleife

Neben den Fallunterscheidungen sind Schleifen ein wesentliches Werkzeug zur Kontrolle des Programmablaufs und damit auch zur Strukturierung eines Programms. Mit Hilfe einer Schleife lassen sich Programmteile wiederholen. Eine der einfachsten Schleifen ist die while-Schleife.

```
while (Ausdruck)
    Anweisung;
```

bzw.

```
while (Ausdruck)
{
    Block von Anweisungen;
}
```

Solange der Ausdruck ungleich 0 ist (Bedingung ist erfüllt, z.B.  $x > y$ ), wird die Anweisung (wird der Anweisungsblock) ausgeführt. Die auszuführende Anweisung bzw. der auszuführende Anweisungsblock nennt sich auch Schleifenkörper. Sollte der Ausdruck vor dem ersten Eintritt in die Schleife den Wert 0 haben, wird die Schleife niemals durchlaufen.

Beispiel:

```
x = 10;
while (x > 0)
{
    /* Anweisungen */

    x = x - 1;
}
```

Im obigen Beispiel werden die Anweisungen des Schleifenkörpers 10 mal durchlaufen. Das obige Beispiel macht deutlich, dass zwei Bedingungen gelten müssen:

1. Die Variable  $x$  (Schleifenkontrollvariable) sollte vor dem ersten Eintreten in die Schleife initialisiert sein, hier mit dem Wert 10.
2. Die Variable  $x$  sollte innerhalb des Schleifenrumpfes geändert werden (Reinitialisierung)

Würde zum Beispiel die Reinitialisierung  $x = x - 1$  fehlen, würde die Schleife unendlich oft durchlaufen (das Programm hängt scheinbar).

Das folgende Beispiel soll ein mögliches Anwendungsgebiet einer while-Schleife illustrieren.

...

```
char ch;
int  weiter;
weiter = 1;

while (weiter == 1)
{
    printf("Bitte geben Sie die auszuführende Aktion an.");
    printf("\nZum Einfügen geben Sie bitte e ein");
    printf("\nZum Ändern geben Sie bitte a ein");
    printf("\nZum Löschen geben Sie bitte d ein");
    printf("\nZum Sortieren geben Sie bitte s ein");
    printf("\nZum Beenden des Programms geben Sie bitte q ein");
    printf("\nIhre Eingabe bitte: ");

    ch=getche();

    switch (ch)
    {
        case 'e':
            einfuegen();
            break;

        case 'a':
            aendern();
            break;

        case 'd':
            loeschen();
            break;

        case 'q':
            weiter = 0;
            break;
    }
}
```

Die Reinitialisierung der Schleifenkontrollvariablen `weiter` geschieht hier durch die Eingabe des Benutzers, das Programm zu beenden.

Eine spezielle while-Schleife ist die Unendlich-Schleife. Es gibt bestimmte Anwendungen, in denen eine Unendlich-Schleife sinnvoll sein kann. Eine mögliche Unendlich-Schleife ist die folgende:

```
while (1)
{
    /* Anweisungen */
}
```

Die Anweisungen des Schleifenkörpers werden unendlich oft wiederholt, da der Ausdruck 1 immer größer als 0 ist.



### 5.3.2 for-Schleife

Eine for-Schleife wird häufig dann verwendet, wenn die Anzahl der Schleifendurchgänge von vornherein bekannt ist, d.h. auch nicht durch einen Benutzer geändert werden kann. Obwohl sich auch solche Schleifen als while-Schleifen formulieren lassen, ist die Formulierung als for-Schleife im Allgemeinen übersichtlicher.

```
for (x = 10; x > 0; x = x - 1)
{
    /* Anweisungen */
}
```

Der Initialisierungsteil und der Reinitialisierungsteil befinden sich mit im Schleifenkopf. Allgemein lautet die Formulierung einer for-Schleife:

```
for (I_Ausdruck; B_Ausdruck; R_Ausdruck)
{
    Anweisungen;
}
```

I\_Ausdruck: → Initialisierung der Kontrollvariablen

B\_Ausdruck: → Schleifenbedingung

R\_Ausdruck: → Reinitialisierung der Kontrollvariablen

**Besonderheiten der for-Schleife**

Der I\_Ausdruck, B\_Ausdruck oder R\_Ausdruck müssen nicht vorhanden sein.

**Beispiele:**

```
x = 10;                                /* schlechter Programmierstil */
for (; x > 0; x = x - 1)
{
    /* Anweisungen */
}
for (x = 10; x > 0; ;)                /* schlechter Programmierstil */
{
    /* Anweisungen */
    x = x - 1;
}
for (x = 10; ; x = x - 1)              /* Unendlich-Schleife !!! */
{                                     /* da fehlender B_Ausdruck */
    /* Anweisungen */                /* B_Ausdruck = wahr */
}
```

Soll wirklich eine Unendlich-Schleife programmiert werden, läßt man üblicherweise alle Ausdrücke im Schleifenkopf weg:

```
for (; ;)                             /* Die beiden Semikolon dürfen nicht vergessen werden */
{
    /* Anweisungen */
}
```

### 5.3.3 do-while

Die fußgesteuerte do-while-Schleife ist ähnlich den kopfgesteuerten while-Schleife mit dem Unterschied, dass die Bedingung für ein wiederholtes Durchlaufen der Schleife nicht zum Schleifenbeginn, sondern am Schleifenende abgefragt wird.

```
do
{
    Anweisungen;
} while (Ausdruck);
```

Das bedeutet, dass die Anweisungen im Schleifenkörper mindestens einmal durchlaufen werden.

Beispiel:

```
i = 0;
do
{
    x = x + 7;
    y = y - 1;
    i = i + 1;
} while (i < 10);
```

Jede do-while-Schleife lässt sich immer auch als while-Schleife schreiben (diese Aussage gilt natürlich auch umgekehrt):

Alle Anweisungen außer der Reinitialisierung müßten noch einmal vor Eintritt in die Schleife ausgeführt werden (sie müssen ja mindestens einmal ausgeführt werden).

### 5.3.4 Geschachtelte Schleifen

Schleifen können beliebig tief geschachtelt werden. Dabei können auch unterschiedliche Typen von Schleifen geschachtelt werden.

Beispiel:

```
x = 0;
for (i = 0; i < 10; i = i + 1)
{
    for (j = 0; j < 5; j = j + 1)
    {
        while (x < 27)
        {
            x = x + 1;
        }
    }
}
```

Nach Beendigung der geschachtelten Schleifen hätte x den Wert 27.

### 5.3.5 break-Anweisung

Eine break-Anweisung kann dazu verwendet werden, um Schleifen oder switch-Anweisungen vorzeitig zu verlassen. Sobald das Programm innerhalb einer switch-Anweisung oder innerhalb einer Schleife auf ein break trifft, wird die switch-Anweisung bzw. die Schleife verlassen.

Beispiel für eine Schleife:

```
for (i = 0; i < 10; i = i + 1)
{
    ia = ia + ib;
    if (ia > 10)
        break;
}
```

Im Falle von for-Schleifen, die bei bestimmten Bedingungen (im Beispiel:  $ia > 10$ ) vorzeitig verlassen werden sollen, ist es immer überlegenswert und in jedem Falle möglich, die Schleife als while- oder do-while-Schleife ohne break zu formulieren.

Ein häufiges Anwendungsfeld für Schleifen mit break sind Unendlich-Schleifen, aus denen man bei Vorliegen einer bestimmten Bedingung rausspringt.

### 5.3.6 continue-Anweisung

Eine continue-Anweisung kann dazu verwendet werden, um im Falle von Schleifen vorzeitig an den Schleifenanfang zu springen. Sobald das Programm innerhalb einer Schleife auf ein continue trifft, wird augenblicklich an den Anfang der Schleife zurückgesprungen.

Beispiel:

```
for (i = 0; i < 10; i = i + 1)
{
    if (i == 7)
        continue;
    printf("\n%d", i);
}
```

Dieser Programmteil schreibt alle ganzen Zahlen von 0 bis 9 mit Ausnahme der 7 auf den Bildschirm.

## 6 Ausdrücke und Operatoren

### 6.1 Ausdrücke

Ein Ausdruck besteht in C aus Operanden und i.a. Operatoren. Jeder Ausdruck hat einen Wert und einen Typ. Beispiele für Ausdrücke sind:

- 1.)  $x + 1$
- 2.)  $(x / y) * (a + b)$
- 3.)  $x < y$
- 4.)  $x = x + 1$
- 5.)  $x$
- 6.)  $3$

Die letzten zwei Beispiele zeigen, dass ein Ausdruck nicht notwendigerweise Operatoren besitzen muss. Auch eine Variable oder Konstante ist ein Ausdruck.

Komplexe Ausdrücke lassen sich aus einfachen Ausdrücken bilden, siehe Beispiel 2. Auch eine Zuweisung, im Beispiel  $x = x + 1$ , ist ein Ausdruck (eine Anweisung würde erst daraus entstehen, wenn dieser Ausdruck mit einem Semikolon abgeschlossen würde).

**Aufgabe:** Welche Werte haben die Ausdrücke 1..6, wenn  $x=6$ ,  $y=2$ ,  $a=3$ ,  $b=4$ ?

- Lösung:**
- 1.) 7
  - 2.) 21
  - 3.) 0 (falsch)
  - 4.) 7
  - 5.) 6
  - 6.) 3

Alle Ausdrücke sind vom Typ Integer, da alle beteiligten Größen von diesem Typ sind.

Um den **Typ eines Ausdrucks** zu bestimmen, gilt folgende Regel:

Haben alle Operanden denselben Typ, hat auch der Ausdruck diesen Typ. Haben die Operanden unterschiedliche Typen, gilt Folgendes:

ein Operand vom Typ *long double*  $\Rightarrow$  Ausdruck vom Typ *long double*  
ansonsten

ein Operand vom Typ *double*  $\Rightarrow$  Ausdruck vom Typ *double*  
ansonsten

ein Operator vom Typ *float*  $\Rightarrow$  Ausdruck vom Typ *float*  
ansonsten

ein Operator vom Typ *unsigned long*  $\Rightarrow$  Ausdruck vom Typ *unsigned long*  
ansonsten

ein Operator vom Typ *long*  $\Rightarrow$  Ausdruck vom Typ *long*  
ansonsten

ein Operator vom Typ *unsigned int*  $\Rightarrow$  Ausdruck vom Typ *unsigned int*  
ansonsten

Ausdruck vom Typ *int*

## 6.2 Operatoren

### 6.2.1 Arithmetische Operatoren

- + Addition
- Subtraktion
- \* Multiplikation
- / Division
- % Rest einer Integer-Division (Modulo)
- arithmetische Negation (z.B. -4)

Bemerkungen:

Punktrechnung geht vor Strichrechnung, Negation hat höhere Priorität als Punktrechnung, Prioritäten können durch runde Klammern beeinflusst werden

Division: Falls ganzzahlige Typen  $\Rightarrow$  Rest wird abgeschnitten (Truncation)

Modulo: Rest einer Division, nur für ganzzahlige Typen

#### Beispiel:

```
void main(void)
{
    int    ix=5, iy=3;
    float  fx=5.0, fy=3.0, fa, fb=2.0;

    fa = fx / fy + fb;    /* fa = 3.6666... */
    fa = ix / iy + fb;    /* fa = 3.0 !!! */
}
```

Der Ausdruck  $ix / iy + fb$  hat den Wert 3.0, denn der Teilausdruck  $ix / iy$  hat den Wert 1 und den Typ *int* (5 geteilt durch 3 ist 1 Rest 2, da Ergebnistyp *int*, wird der Rest abgeschnitten). Der gesamte Ausdruck  $ix / iy + fb$  hat den Typ *float* (Regel: Teilausdruck hat den Typ *int*, *fb* hat den Typ *float*  $\Rightarrow$  Ausdruck hat den Typ *float*).

#### HINWEIS:

Haben Operatoren dieselbe Priorität, wird von links nach rechts ausgewertet.

$4 * 5 / 2$  hat den Wert 10

bei Auswertung von rechts nach links hätte  $4 * 5 / 2$  den Wert 8, denn  $5 / 2$  ist 2

Bei der Verwendung von Konstanten kann der Datentyp durch Anhängen eines Kennbuchstaben erzwungen werden:

L	$\rightarrow$	long int
U	$\rightarrow$	unsigned int
F	$\rightarrow$	float

#### Beispiele:

Ausdruck	Wert
$3 + 3 * 4 / 2$	9
$3 + 3 / 2 * 4$	7
$3 + 3 / 2.0 * 4$	9.0
$3 + 3 / 2F * 4$	9.0
$3 + 3 \% 2 * 4$	7
$3 + 3 * 4 \% 2$	3
$(3 + 3) \% 2 * 4$	0
$(3 + 3) \% (2 * 4)$	6
$(10 + 1) / (4 \% 2)$	Division durch 0 !!!

## 6.2.2 Vergleichsoperatoren

== Überprüfen auf Gleichheit  
!= Überprüfen auf Ungleichheit  
< Überprüfen auf kleiner  
<= Überprüfen auf kleiner oder gleich  
> Überprüfen auf größer  
>= Überprüfen auf größer oder gleich

Vergleichsoperatoren werden zum Vergleich von Operanden benutzt. Das Ergebnis einer Vergleichsoperation ist logisch wahr oder logisch falsch. Bei logisch wahr hat der entsprechende Ausdruck den Wert 1, bei logisch falsch den Wert 0.

### Beispiele:

```
int    ix, ia=3, ib=8;
float  fy, fa=4.2, fb=2.3;
ix = ia >= ib;          /* ix = 0 */
fy = (fa != fb) + 1;    /* fy = 2.0 */
fy = fa != fb + 1;      /* fy = 1.0 */
```

Die Priorität von Vergleichsoperatoren ist geringer als die von arithmetischen Operatoren (deshalb die runden Klammern in der vorletzten Zeile des Beispiels). Innerhalb der Gruppe der Vergleichsoperatoren haben die Operatoren == und != eine geringere Priorität als die restlichen Operatoren.

### ACHTUNG:

Es ist, insbesondere bei if-Anweisungen darauf zu achten, dass der Vergleichsoperator == nicht mit dem Zuweisungsoperator = verwechselt wird.

Aus `if (x==4)` kann fälschlicherweise leicht ein `if (x=4)` werden, was zu dem Ergebnis führt, dass die Bedingung immer logisch wahr ist, denn der Ausdruck `x = 4` hat immer den Wert 4 und ist daher ungleich 0.

## 6.2.3 Logische Operatoren

&& logische UND-Verknüpfung  
|| logische ODER-Verknüpfung  
! logische Negation

Logische Operatoren werden häufig im Zusammenhang mit Vergleichsoperatoren benutzt.

### Beispiele:

```
if ((x < y) && (x >= 0)) . . .
if ((x < y) || !(a < b)) . . .
```

Der logische Ausdruck im ersten if ist nur dann 1 (logisch wahr), wenn sowohl `x < y` als auch `x >= 0` gilt. Der Ausdruck im zweiten if ist nur dann 1, wenn entweder `x < y` oder nicht `a < b` gilt.

Die Auswertungsreihenfolge ist von links nach rechts, aber nur solange bis das Ergebnis feststeht. Wenn in der zweiten if-Anweisung `x` kleiner als `y` wäre, würde der zweite Teilausdruck `!(a < b)` nicht mehr ausgewertet werden. Das Ergebnis des gesamten Ausdrucks wäre schon nach Auswerten von `(x < y)` klar.

### Prioritäten:

Die logische Negation hat die gleiche Priorität wie die arithmetische Negation. Die beiden anderen logischen Operatoren haben geringere Priorität als Vergleichsoperatoren. Die if-Anweisung

```
if ((x < y) && (x >= 0)) . . .
```

ist äquivalent der Variante ohne Klammern

```
if (x < y && x >= 0) . . .
```

da wegen der höheren Priorität der Vergleichsoperatoren gegenüber den logischen Operatoren erst  $x < y$  ausgewertet, dann  $x \geq 0$  ausgewertet und anschließend die logische UND-Verknüpfung durchgeführt wird.

## 6.2.4 Bitoperatoren

& bitweise UND-Verknüpfung  
| bitweise ODER-Verknüpfung  
^ bitweise exklusive ODER-Verknüpfung  
~ Komplement  
>> Rechtsshift  
<< Linksshift

Bitoperatoren können nur auf integrale Typen angewendet werden und operieren auf jedem Bit von Variablen (bzw. Konstanten) einzeln. Die Operatoren ~, >> und << arbeiten nur auf einer Größe, während die Operatoren &, | und ^ zwei Größen miteinander verknüpfen.

### Wahrheitstafel:

bit1	bit2	~bit1	bit1 & bit2	bit1   bit2	bit1 ^ bit2
0	0	1	0	0	0
0	1	1	0	1	1
1	0	0	0	1	1
1	1	0	1	1	0

### Beispiele:

**x << 4**

schiebt x um 4 Bit nach links und füllt von rechts mit Nullen auf  
entspricht einer Multiplikation mit  $2^n$

z.B.  $x = 15$  (Bitmuster: 00001111)  $\Rightarrow x \ll 4$  hat den Wert 240 (11110000)

**x >> 4**

schiebt x um 4 Bit nach rechts

x unsigned  $\Rightarrow$  Auffüllen von links mit Nullen

x signed  $\Rightarrow$  Auffüllen von links mit Einsen oder Nullen (implementierungsabhängig)

$x = 16$  (Bitmuster: 00010000)  $\Rightarrow \sim x$  hat Bitmuster 11101111

$x = 15$  (00001111),  $y = 4$  (00000100)

**x & y** hat Bitmuster 00000100  $\Rightarrow$  Wert = 4

**x | y** hat Bitmuster 00001111  $\Rightarrow$  Wert = 15

**x ^ y** hat Bitmuster 00001011  $\Rightarrow$  Wert = 11

Bitweise UND- und ODER-Verknüpfungen werden eingesetzt, um gezielt Bits zu setzen oder zu löschen (z.B. zum Programmieren von Ports oder Registern).

Zum gezielten Löschen von einzelnen Bits (Setzen auf 0) in einer Variablen ist die entsprechende Variable UND zu verknüpfen mit einer Konstanten, bei der alle Bits auf 1 gesetzt sind mit Ausnahme der Bits, die in der Variablen gelöscht werden sollen.

**Beispiel:** Löschen des 1. und 8. Bits in  $x \Rightarrow x = x \& 126;$  ( $126 = 01111110$ )

Zum gezielten Setzen von einzelnen Bits (Setzen auf 1) in einer Variablen ist die entsprechende Variable ODER zu verknüpfen, und zwar mit einer Konstanten, bei der alle Bits auf 0 gesetzt sind mit Ausnahme der Bits, die in der Variablen gesetzt werden sollen.

**Beispiel:** Setzen des 4. und 5. Bits in  $y \Rightarrow y = y \mid 24;$  ( $24 = 00011000$ )

## 6.2.5 Zuweisungsoperatoren

=	z.B. $y = x + 4$	
+=	z.B. $x += 4$	$\Leftrightarrow x = x + 4$
-=	z.B. $z -= x$	$\Leftrightarrow z = z - x$
*=	z.B. $x *= 7$	$\Leftrightarrow x = x * 7$
/=	z.B. $x /= 5$	$\Leftrightarrow x = x / 5$
%=	z.B. $x \% = 3$	$\Leftrightarrow x = x \% 3$
>>=	z.B. $x >> = 3$	$\Leftrightarrow x = x >> 3$
<<=	z.B. $y << = z$	$\Leftrightarrow y = y << z$
&=	z.B. $y \& = 126$	$\Leftrightarrow y = y \& 126$
=	z.B. $z \mid = 17$	$\Leftrightarrow z = z \mid 17$
^=	z.B. $x \wedge = a$	$\Leftrightarrow x = x \wedge a$
++	z.B. $x++$	$\Leftrightarrow x = x + 1$
--	z.B. $y--$	$\Leftrightarrow y = y - 1$

Von all den Zuweisungsoperatoren finden sich in der Praxis im Wesentlichen drei wieder, nämlich =, ++ und --.

Die Operatoren ++ und -- lassen sich entweder als Postfix oder als Präfix anwenden.

Postfix:  $y = x++;$   $\Rightarrow$  erst Zuweisung von x zu y, dann Erhöhung von x um 1.

Präfix:  $y = ++x;$   $\Rightarrow$  erst Erhöhung von x um 1, dann Zuweisung des erhöhten x zu y.

## 6.2.6 Übrige Operatoren

### Bedingungsoperator

Ausdruck1 ? Ausdruck2 : Ausdruck3

Ausdruck1 ungleich 0 (logisch wahr)  $\Rightarrow$  Auswertung von Ausdruck2

Ausdruck1 gleich 0 (logisch falsch)  $\Rightarrow$  Auswertung von Ausdruck3

z.B.  $(ix \neq iy) ? (iz = 5) : (iz = 0);$

```
entspricht  if (ix != iy)
              iz = 5;
              else
              iz = 0;
```

### Sequenzoperator (Kommaoperator)

ermöglicht, zwei Ausdrücke als einen aufzufassen

z.B.  $x++; y++;$  entspricht  $x++, y++;$



## Größenoperator

ermittelt den Speicherbedarf einer Variablen oder eines Datentyps in Byte

```
z = sizeof(int);  
y = sizeof(x);
```

## Adressoperator

Variable  $x \Rightarrow \&x$  ist die Speicheradresse von  $x$

benutzt z.B. bei `scanf()`

## Cast-Operator

benutzt zur expliziten Typumwandlung

Beispiele:

```
float fz;  
int ix=3, iy=2;  
fz = ix / iy;           /* fz = 3 / 2 = 1.0 */  
fz = (float) ix / iy;   /* fz = 3.0 / 2 = 1.5 */  
fz = (float) (ix / iy); /* fz = 3 / 2 = 1.0 */
```

Im ersten Beispiel werden bei der Division zweier Ganzzahl die Nachkommastellen abgeschnitten und dann erst das Ergebnis der Variablen `fz` zugewiesen (Rundungsfehler).

Im zweiten Beispiel wird die linke Seite des Ausdrucks `ix / iy` explizit in den Typ *float* umgewandelt. Damit wird eine Gleitkommazahl (3.0) durch eine Ganzzahl (2) geteilt, das Ergebnis ist wiederum eine Gleitkommazahl (1.5). Dieses Ergebnis wird der Variablen `fz` zugewiesen.

Im dritten Beispiel wird zuerst eine Division zweier Ganzzahlen durchgeführt mit dem Ergebnis 1, dann wird dieses Ergebnis explizit in den Datentyp *float* umgewandelt (1.0) und danach der Variablen `fz` zugewiesen (nicht das, was man eigentlich haben wollte).

## 6.2.7 Prioritäten von Operatoren

Operatoren	Auswertung
<code>() [] -&gt; .</code>	von links
<code>! ~ ++ -- + - * &amp; (type) sizeof</code>	von rechts
<code>* / %</code>	von links
<code>+ -</code>	von links
<code>&lt;&lt; &gt;&gt;</code>	von links
<code>&lt; &lt;= &gt; &gt;=</code>	von links
<code>== !=</code>	von links
<code>&amp;</code>	von links
<code>^</code>	von links
<code> </code>	von links
<code>&amp;&amp;</code>	von links
<code>  </code>	von links
<code>? :</code>	von rechts
<code>= += -= *= /= %= &amp;= ^=  = &lt;&lt;= &gt;&gt;=</code>	von rechts
<code>,</code>	von links

## 7 Funktionen

Grundsätzlich sollten größere Programme in übersichtliche Teile zerlegt werden. Diese Teile können dann immer noch so komplex sein, dass sie weiter unterteilt werden usw. Der Entwurf dieser Hierarchie nennt sich Prinzip der schrittweisen Verfeinerung. (TOP-DOWN Design)

Diese Teilprogramme werden Funktionen genannt. Eine Funktion erledigt eine abgeschlossene Teilaufgabe. Weiterhin müssen der Funktion die zu verarbeitenden Werte mitgegeben werden, das sind sogenannte Argumenten bzw. Parametern. Die Funktion liefert das Ergebnis an die aufrufende Funktion zurück = Rückgabewert. Ein Beispiel ist die Sinus-Funktion  $y = \sin(x)$ : Hier wird der Wert  $x$  der Funktion als Parameter übergeben und das Ergebnis der Sinus-Funktion wird der Variablen  $y$  zugewiesen.

Eine Funktion muss nur einmal definiert werden. Danach kann sie beliebig oft durch Nennung ihres Namens (dem Funktionsnamen) aufgerufen werden. Eine Funktion ist also wiederverwendbar!

### 7.1 Funktionsprototypen und -definitionen

Bei Funktionen wird zwischen der Deklaration (Bekanntmachung des Namens) und der Definition (Belegung eines Speicherbereichs) unterschieden. Die Deklaration der Funktionen wird im allgemeinen vor dem Hauptprogramm `main()` geschrieben. Diese Deklarationen werden Funktionsprototypen genannt. Die Syntax eines Funktionsprototypen sieht folgendermaßen aus:

```
Rückgabe-Datentyp Funktionsname(Datentyp Parametername, ...);
```

Die Parameter werden bei Funktionsprototypen und -definitionen auch Formalparameter genannt. Die Parameteranzahl kann bei jeder Funktion unterschiedlich sein. Es können Funktionen mit einer Parameteranzahl zwischen 0 und "beliebig viele" deklariert und definiert werden. Beim Aufruf der Funktion dagegen muss exakt die bei der Deklaration bzw. der Definition vorgegebene Anzahl der Parameter angegeben werden.

Durch die Deklaration "weiß" der Compiler, dass irgendwo eine Funktion mit dem angegebenen Funktionsnamen definiert ist, wieviele Parameter sie hat und welchen Datentyp der Rückgabewert hat. Wird diese Funktion irgendwo im Programm aufgerufen, kann der Compiler jetzt eine Syntaxprüfung durchführen, ohne dass die Funktion definiert sein muss.

Erst durch die Funktionsdefinition kann die Funktion auch angewendet werden, denn erst hier wird der Quellcode angegeben und wird entsprechend Speicherplatz dafür reserviert. Die Syntax einer Funktionsdefinition sieht folgendermaßen aus:

```
Rückgabe-Datentyp Funktionsname(Datentyp Parametername, ...)
{
    Anweisungen;
}
```

Wichtigster Unterschied in der Syntax zwischen Funktionsprototyp und -definition ist: der Prototyp ist mit einem Semikolon abgeschlossen! Dafür folgen bei der Definition in den darauffolgenden Zeilen der Quellcode der Funktion. Der Quellcode der Funktion wird - auch

wenn die Funktion nur eine Anweisung beinhaltet - zwischen einem Paar geschweifte Klammern gesetzt.

**Beispiel:**

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

// Funktionsprototyp:
double Average(double Zahl1, double Zahl2, double Zahl3);

// Hauptprogramm:
int main(void)
{
    double a = 4.5, b = 3.1415, c = 7.99;

    // Aufruf der Funktion in der printf-Anweisung:
    printf("Durchschnitt: %f", Average(a,b,c));
    return EXIT_SUCCESS;
}

// Funktionsdefinition:
double Average(double Zahl1, double Zahl2, double Zahl3)
{
    return (Zahl1 + Zahl2 + Zahl3) / 3;
}
```

Aufgerufen wird die Funktion durch den Funktionsnamen gefolgt den Parametern, die in Klammern gesetzt werden. Die Klammern müssen auch dann gesetzt werden, wenn keine Parameter der Funktion übergeben werden.

Die Syntax für einen Funktionsaufruf lautet also

```
Funktionsname(Variablenname bzw. Konstante, ...);
```

Der Rückgabewert der Funktion kann in einer Variablen gespeichert werden gleich einer anderen Funktion übergeben werden.

### **7.1.1 Gültigkeitsbereiche und Sichtbarkeit**

Grundsätzlich sind alle deklarierten Namen von Variablen, Typen, Konstanten, Funktionen, usw. nach der Deklaration nur innerhalb des Blocks gültig, in dem sie deklariert wurden. Diese Variablen werden lokale Variablen genannt; sie sind nur in der lokalen Umgebung (innerhalb des Blocks) bekannt. Nach Verlassen des Blocks werden sie wieder vernichtet, d.h. ihr Speicherplatz wird wieder freigegeben.

Variablen, die außerhalb von allen Blöcken deklariert werden, sind innerhalb der ganzen Datei gültig, d.h. innerhalb der main- und innerhalb aller anderen Funktionen, die in der gleichen Quellcodedatei definiert sind.

### Beispiel:

```
int a; // globale Variable

void Test(void); //keine Parameter, kein Rückgabewert

void main()
{   int b; // lokal im Hauptprogramm
    a = 0; // erlaubt, da a global
    b = 0; // erlaubt, da b in diesem Block deklariert
    c = 0; // FEHLER!, da c nur in Funktion bekannt
    Test();
}

void Test(void)
{   int c; // lokal in dieser Funktion
    a = 0; // erlaubt, da a global
    b = 0; // FEHLER!, da b nur im Hauptprogramm bekannt
    c = 0; // erlaubt, da c in dieser Funktion deklariert
}
```

**Auf globale Variablen sollte nach Möglichkeit verzichtet werden und statt dessen lieber diese Variablen lokal angelegt und bei Bedarf als Parameter an die Funktionen übergeben werden.**

Eine Ausnahme bilden lokale Variable, die innerhalb eines Blocks oder einer Funktion als statische Variable definiert werden. Dazu wird vor dem Datentyp zusätzlich das Schlüsselwort `static` verwendet. Diese statischen Variablen werden nach Verlassen des Blocks oder der Funktion nicht vernichtet und haben bei erneuten Aufruf des Block oder der Funktion noch den alten Wert. Ferner wird eine evtl. Variableninitialisierung nur bei der erstmaligen Definition ausgeführt; bei allen weiteren Aufrufen wird die Initialisierung ignoriert.

### Beispiel:

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

void Test(void);

int main()
{   for (int i = 0; i < 3; i++)
        Test();
    Return EXIT_SUCCESS;
}

void Test(void)
{   static int Anzahl = 0;
    // wird nur beim 1. Aufruf auf Null gesetzt!
    Anzahl++;
    printf("\n Anzahl = %i",Anzahl);
}
```

Die Ausgabe des Programms ist

```
Ausgabe = 1
Ausgabe = 2
Ausgabe = 3
```

Ohne das Schlüsselwort `static` würde dreimal eine 1 ausgegeben werden, da die lokale Variable bei jedem Funktionsaufruf wieder neu erzeugt würde.

## 7.1.2 Funktionsschnittstelle

Unter einer Schnittstelle ist eine formale Vereinbarung zwischen Aufrufer und Funktion über die Art und Weise des Datentransports zu verstehen. Auch das, was die Funktion leistet, gehört zur Schnittstelle (sollte als Kommentar beim Funktionskopf stehen). Die Schnittstelle ist durch den Funktionsprototyp eindeutig beschrieben und enthält folgendes:

- den Rückgabebetyp der Funktion,
- den Funktionsnamen,
- Parameter, die der Funktion bekannt gemacht werden inkl. deren Datentypen und
- die Art der Parameterübergabe.

Der Compiler prüft, ob die Definition der Schnittstelle bei einem Funktionsaufruf eingehalten wird.

Für den Datentransfer in die Funktion hinein gibt es verschiedene Arten des Datentransports:

### 7.1.3 Übergabe per Wert

Bei der Übergabe per Wert wird der Wert erst kopiert und dann die Kopie der Funktion übergeben. Innerhalb der Funktion wird also nur mit einer Kopie gearbeitet, die am Ende der Funktion wieder vernichtet wird, während das Original unverändert bleibt.

Beispiel:

```
#include <stdio.h>

int Addiere_3(int x);

void main()
{   int Ergebnis, Zahl = 2;

    printf("\n\nWert von Zahl = %i", Zahl);
    Ergebnis = Addiere_3(Zahl);
    printf("\n\nErgebnis 'Addiere_3(Zahl)' = %i", Ergebnis);
    printf("\n\nWert von Zahl = %i (unverändert)", Zahl);
}

int Addiere_3(int x)
{   x += 3;
    return x;
}
```

Die Übergabe per Wert sollte generell verwendet werden, wenn ein Objekt von der Funktion nicht verändert werden soll.

### 7.1.4 Übergabe per Referenz

Soll ein übergebenes Objekt von der Funktion modifiziert werden können, wird die Übergabe per Referenz verwendet. Die Syntax für den Aufruf ist die gleiche wie bei der Übergabe per Wert, allerdings wird hier innerhalb der Funktion direkt mit dem Original gearbeitet, d.h. nach Beenden der Funktion kann das Originalobjekt einen anderen Wert haben.

Der einzige Unterschied zwischen der Übergabe per Wert und per Referenz liegt bei den Datentypen der Parameter (bei Funktionsprototypen und -definitionen): Für eine Übergabe per Referenz wird an den Datentypen des Parameters oder vor dem Parameter ein kaufmännisches Und ("&") angehängt (im Beispiel: `int& x` anstelle von `int x`).

Beispiel:

```
#include <stdio.h>

int Addiere_3(int& x);

void main()
{   int Ergebnis, Zahl = 2;

    printf("\n\nWert von Zahl = %i", Zahl);
    Ergebnis = Addiere_3(Zahl);
    printf("\n\nErgebnis 'Addiere_3(Zahl)' = %i", Ergebnis);
    printf("\n\nWert von Zahl = %i (veraendert!!!)", Zahl);
}

int Addiere_3(int& x)
{   x += 3;
    return x;
}
```

### 7.1.5 Übergabe per Zeiger

Die Übergabe per Zeiger ist ein Spezialfall der Übergabe per Wert, es wird nämlich die Adresse eines Objekt übergeben. Innerhalb der Funktion kann über diesen Zeiger auf das eigentliche Objekt zugegriffen (und damit auch verändert) werden. Es wird also nicht das Objekt selber übergeben, sondern ein Zeiger auf das Objekt.

Bei den Parametern wird vor dem Parameternamen ein Stern gesetzt, da ja ein Zeiger übergeben wird; der Datentyp dagegen wird nicht geändert. Beim Aufruf selber wird vor dem Parameternamen ein kaufmännisches Und ("&") gesetzt. Dadurch wird ein Zeiger auf das eigentliche Objekt erzeugt und der Funktion übergeben. Das obige Beispiel wird nun so geändert, dass die Zahl per Zeiger an die Funktion übergeben wird.

Beispiel:

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

int Addiere_3(int *x);

int main()
{   int Ergebnis, Zahl = 2;

    printf("\n\nWert von Zahl = %i", Zahl);
    Ergebnis = Addiere_3(&Zahl);
    printf("\n\nErgebnis 'Addiere_3(&Zahl)' = %i", Ergebnis);
    printf("\n\nWert von Zahl = %i (verändert!!!)", Zahl);
    return EXIT_SUCCESS;
}

int Addiere_3(int *x)
{   *x += 3;
    return *x;
}
```

### 7.1.6 Rückgabewerte

Der Rückgabewert wird innerhalb der Funktion mit dem Befehl `return Wert;` angegeben, wobei Wert der Rückgabewert ist. Gleichzeitig beendet dieser Befehl die Funktion, unabhängig davon, ob weitere Anweisungen folgen oder nicht.

Beispiel:

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

int Addiere_3(int x);

int main()
{   int Ergebnis, Zahl = 2;

    printf("\n\nWert von Zahl = %i", Zahl);
    Ergebnis = Addiere_3(Zahl);
    printf("\n\nErgebnis 'Addiere_3(Zahl)' = %i", Ergebnis);
    printf("\n\nWert von Zahl = %i (unverändert)", Zahl);
    return EXIT_SUCCESS;
}

int Addiere_3(int x)
{   x += 3;
    return x;
    printf("STOPP!"); // Diese Anweisung wird nie ausgeführt!
}
```

### 7.1.7 Rekursiver Funktionsaufruf

Der Aufruf einer Funktion durch sich selbst wird Rekursion genannt. Eine Rekursion muss irgendwann auf eine Abbruchbedingung stoßen, damit die Rekursion nicht unendlich ist. Bei einer unendlichen Rekursion wird irgendwann ein sogenannter Stacküberlauf (stack overflow) erzeugt; das Programm wird damit abgebrochen.

*Als Beispiel für die Verwendung einer Rekursion wird die Quersumme einer ganzen Zahl berechnet: Eine Funktion ermittelt die letzte Ziffer einer Zahl, addiert diese zur Quersumme und ruft sich selbst mit den restlichen Ziffern wieder auf.*

Beispiel: Ziffernsumme

Prinzip:

1. Die Quersumme der Zahl 0 ist gleich 0. Dies ist die Abbruchbedingung für die Rekursion!
2. Die Quersumme einer Zahl ist gleich der letzten Ziffer plus der Quersumme der Zahl, die um diese Ziffer gekürzt wurde.

Die Quersumme von 873956 ist also gleich 6 plus der Quersumme von 87395 und ist damit gleich 6 plus 5 plus der Quersumme von 8739 usw. Auf jede Quersumme wird der Satz 2 angewandt, bis der Satz 1 gilt. Satz 1 gilt dann, wenn alle Ziffern von der Zahl abgetrennt wurden.

### Beispiel:

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

int Quersumme(unsigned long x);

int main()
{   unsigned long Zahl;

    printf("\nBitte eine positive ganze Zahl eingeben: ");
    scanf("%lu",&Zahl);
    printf("\nQuersumme von %lu ist %i",Zahl,Quersumme(Zahl));
    return EXIT_SUCCESS;
}

int Quersumme(unsigned long x)
{   int letzteZiffer = 0;

    if (x) // if (x == 0) Abbruchbedingung;
    {   letzteZiffer = x % 10; // modulo 10
        return letzteZiffer + Quersumme(x / 10); // Rekursion
    }
    else
        return 0; // Abbruch der Rekursion
}
```

Die Funktion Quersumme kann auch nicht-rekursiv geschrieben werden, in dem eine Schleife verwendet wird. Diese Variante wird iterativ genannt.

```
int Quersumme(unsigned long x)
{   int QSumme = 0;

    while (x > 0)
    {   QSumme += x % 10; // letzte Ziffer addieren
        x /= 10; // letzte Ziffer abtrennen
    }
    return QSumme; // Rückgabe der Quersumme
}
```

## 7.2 Die Funktion main()

Die Funktion main() - das Hauptprogramm - ist eine spezielle Funktion. Jedes C-/C++-Programm startet definitionsgemäß mit main(), so dass main() in jedem Programm genau einmal vorhanden sein muss.

Einfache Form:

```
int main(){
    //...
    return EXIT_SUCCESS; // Exit-Code
}
```



## 8 Felder und Strings

### 8.1 Felder

#### 8.1.1 Eindimensionale Felder

Ein eindimensionales Feld entspricht einer eindimensionalen Tabelle, in der eine Anzahl von Variablen gleichen Datentyps gespeichert werden kann. Die einzelnen Variablen liegen hintereinander im Speicher.

Eindimensionale Felder werden üblicherweise benutzt, um zusammengehörige Variablen in einer gemeinsamen Datenstruktur zu speichern.

Beispiel Aktienkurse der letzten 30 Tage:

```
float kurs1, kurs2, kurs3, . . ., kurs29, kurs30;
```

**alternativ:**

```
float kurs[30];
```

Mit obiger Definition wird eine Feldvariable definiert, die aus 30 Feldelementen besteht, die wiederum Variablen sind, in diesem Fall vom Typ float. Auf die einzelnen Feldelemente läßt sich über einen Index zugreifen. Das erste Feldelement hat immer den Index 0, das letzte Feldelement im obigen Beispiel den Index 29.

Will man zum Beispiel dem ersten Feldelement den Wert 17.42 zuweisen, lautet die entsprechende Anweisung:

```
kurs[0] = 17.42;
```

Will man überprüfen, ob das fünfte Feldelement größer ist als 100, lautet die entsprechende Anweisung:

```
if (kurs[4] > 100) /* das 5. Feldelement hat den Index 4 */
```

Elemente von Feldern lassen sich genauso benutzen wie normale Variablen. Feldelemente können jeden Datentyp annehmen, natürlich aber alle Elemente einer Feldvariablen nur denselben.

Sollen für alle Aktienkurse Werte von der Tastatur eingelesen werden, erreicht man dies sehr einfach durch die Anweisung:

```
for (i=0; i<30; i++)
{
    printf("\nBitte geben Sie den Kurs fuer die Aktie %d ein: ", i);
    scanf("%f", &kurs[i]);
}
```

Auch Felder lassen sich, wie einfache Variablen auch, im Rahmen der Definition initialisieren.

```
float kurs[10] = {1, 2, 3, 3, 2, 4.5, 3.1, 3.1, 1, 0 };
```

kurs[0] wäre in diesem Fall 1, kurs[1] wäre 2 u.s.w.

Wird nur ein Teil der Initialwerte angegeben,

```
float kurs[10] = {1, 2, 3, 3};
```

werden die restlichen Elemente auf 0 gesetzt.

Um zum Beispiel alle Elemente des Feldes bei der Definition mit 0 zu initialisieren, würde reichen:

```
float kurs[10] = {0};
```

### Komplexeres Beispiel:

*/\* Haeufigkeitsverteilung von Vokalen in einer Eingabe \*/*

```
#include <stdio.h>
int main(void) {

    char ch=' ';
    int iH[5]={0},i;
    printf("Das Programm zaehlt die Vokale in Ihrer Eingabe.\n");
    printf("Geben Sie Text ein.\n");
    ch = getchar();
    while (ch != '\n')          // lesen in den Tastaturpuffer bis
    {                            // <ENTER>= '\n' gedrückt wird
        switch (ch)
        {
            case 'A':
            case 'a': iH[0]++;
                     break;

            case 'E':
            case 'e': iH[1]++;
                     break;

            case 'I':
            case 'i': iH[2]++;
                     break;

            case 'O':
            case 'o': iH[3]++;
                     break;

            case 'U':
            case 'u': iH[4]++;
                     break;

        }
        ch = getchar();
    }
    printf("\nHaeufigkeiten der Vokale:\n");
    printf("\na\te\ti\to\tu\n");
    printf("\n%d\t%d\t%d\t%d\t%d", iH[0],iH[1],iH[2],iH[3],iH[4]);
    return(0);
}
```

### ANMERKUNG:

Direkte Zuweisungen des gesamten Feldes sowie Vergleiche von gesamten Feldern sind nicht erlaubt.

```
int feld1[10], feld2[10], i;

for (i=0; i<10; i++)
    feld1[i] = 10;

feld2 = feld1;          /* nicht erlaubt */

for (i=0; i<10; i++)
    feld2[i] = feld1[i]; /* so geht's */

if (feld1 == feld2)     /* nicht erlaubt */

if (feld1[0] == feld2[0] && feld1[1] = feld2[1] && . . .) /* so geht's */
```

## 8.1.2 Mehrdimensionale Felder

Es lassen sich auch mehrdimensionale Arrays definieren. Dabei muss für jede Dimension die Anzahl der Elemente in eckige Klammern angegeben werden, und zwar so, dass jede Dimension ein eigenes Klammerpaar hat, z.B. `Matrix[3][3]`.

Man kann sich ein zweidimensionales Feld als Tabelle (Matrix) vorstellen:  
Bsp: `Matrix[2][3]`

```
int Matrix[2][3]={{0,3,5},{2,1,4}};
```

Matrix	[0]	[1]	[2]	2. Index
[0]	0	3	5	
[1]	2	1	4	
1. Index				

Beispiel: (Beilage)

```
int i,j,Matrix[3][3] = { {1, 2, 3}, {4, 5, 6} };

// Ausgabe der 3*3-Matrix:
for (i = 0; i < 3; i++)
{
    for (j = 0; j < 3; j++)
        printf("%3i ",Matrix[i][j]);
    printf("\n"); // Zeilenumbruch nach jeder Matrixzeile
}
```

## 8.2 Zeichenketten (Strings)

### 8.2.1 Definition und Zuweisung

Ein spezielles eindimensionales Feld ist eine Zeichenkette, die auch als String bezeichnet wird. Ein String ist ein eindimensionales Feld von Zeichen zur Speicherung von Wörtern, Sätzen u.s.w.

```
char s[20];
```

Ein String kann immer ein Zeichen weniger speichern als in der Definition angegeben, im obigen Beispiel also höchstens 19 reguläre Zeichen. Der Grund ist, dass mindestens das letzte Element eines Strings für das Stringendezeichen `'\0'` reserviert bleiben muss. Am Stringendezeichen erkennen bestimmte Operationen das Ende eines Strings. Wegen des Stringendezeichens heißen Strings in C auch nullterminiert.

Die Zuweisung einer Zeichenkettenkonstante oder eines anderen Strings an eine Stringvariable kann nicht über den Zuweisungsoperator geschehen, da einem Feld ja nicht als Ganzes etwas zugewiesen werden kann. Deshalb gibt es Standardfunktionen zur Manipulation von Strings. Auf die Zuweisung bezogen ist dies die Funktion `strcpy()` (String-Copy), mittels der einem String ein Wert zugewiesen werden kann.

```
strcpy(s, "Felix Unger");
```

Der Inhalt der Variablen `s` besteht aus den Zeichen Felix Unger zuzüglich des Stringendezeichens. Das bedeutet, dass z.B. `s[0]` das Zeichen 'F', `s[6]` das Zeichen 'U' und `s[11]` das Zeichen `'\0'` beinhaltet. Die Feldelemente `s[12]` bis `s[19]` haben einen undefinierten Wert.

## 8.2.2 Ein- und Ausgabe

Die Ein- und Ausgabe von Strings muss nicht zeichenweise, sondern kann mittels `scanf()` und `printf()` mit der Formatanweisung `%s` auf einen Schlag erfolgen.

```
scanf("%s", s);    ↔    scanf("%s", &s[0]);  
printf("Der Stringinhalt lautet: %s", s);
```

Dummerweise liest `scanf()` nur bis zum ersten Leerzeichen, so dass bei einer Benutzereingabe

Felix Unger<Enter>

der Stringinhalt nur `Felix\0` wäre. Das nachfolgende `printf()` würde dann auch nur Felix ausgeben. Aus obigem Grunde gibt es neben `scanf()` eine weitere Funktion zum Einlesen von Strings:

```
gets(s);
```

`gets()` liest die Benutzereingabe inklusive <Enter> aus dem Tastaturpuffer und speichert alle Zeichen der Eingabe (inklusive Leerzeichen) in der Stringvariablen, in obigem Beispiel in der Variablen `s`. Das <Enter>-Zeichen `'\n'` wird automatisch vor dem Speichern in der Stringvariablen in das Zeichen `'\0'` umgewandelt.

Eine zu `gets()` korrespondierende Ausgabefunktion ist `puts()`.

```
#include <stdio.h>  
int main(void)  
{  
    char str[25];  
    gets(str);  
    /* Eingabe: Felix Unger<Enter> */  
    puts(str);  
    /* schreibt Felix Unger auf den Bildschirm */  
    return(0);  
}
```

Anstelle der Ausgabe mittels `puts()` hätte man auch `printf()` benutzen können:  
`printf("%s", str);`

## 8.2.3 Initialisierung von Strings

Wie alle anderen Variablen auch können Strings direkt im Rahmen ihrer Definition mit einer Stringkonstanten initialisiert werden.

```
char str[20] = "Felix Unger";
```

**ANMERKUNG:**

Das Gleichheitszeichen darf nur zur Zuweisung eines Initialwertes im Rahmen der Variablendefinition benutzt werden. Zuweisungen im Anweisungsteil des Programms dürfen nur mittels der Funktion `strcpy()` erfolgen.

## 8.2.4 Funktionen zur Stringmanipulation

Es gibt in C eine Menge von Standardfunktionen zur Stringmanipulation. Einige wichtige Funktionen sind:

```
strcpy(str1, str2);
```

kopiert den String (oder die Stringkonstante) str2 in den String str1 (Zuweisung)

```
strcat(str1, str2);
```

hängt den String (oder die Stringkonstante) str2 an den String str1 an (str1 muss groß genug definiert werden !!!)

```
strcmp(str1, str2);
```

vergleicht zwei Strings miteinander (Rückgabewert der Funktion ist 0, wenn beide Strings den gleichen Inhalt haben)

Beispiel:

```
if (strcmp(str1, str2) == 0)
```

```
strlen(str);
```

Rückgabewert der Funktion ist die Länge des Strings (Typ: int)

Beispiel:

```
len = strlen(s);
```

Um die Funktionen zur Stringmanipulation dem Programm bekannt zu machen, muss zu Beginn des Programms die Programmzeile

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

eingefügt werden.

## Zeichenweise Stringmanipulationen

Anstelle von `gets(str)`; kann man auch elementar schreiben:

```
i = -1;
do
{
    i++;
    str[i] = getchar();
}
while (str[i] != '\n');
str[i] = '\0';
```

Die obigen Beispiele machen wohl deutlich, dass Stringmanipulationen oder Ein- bzw. Ausgabe von Strings besser über entsprechende Funktionen durchzuführen sind.

### Beispiel: Passworteingabe

```
#include <string.h>
#include <stdio.h>
#include <conio.h>      /* kein ANSI-Standard */
int main(void)
{
    char cBuffer[81], cPWort[9], cZeig, cNeu, c;
    do
    {
        printf("Passwort eingeben (max. 8 Zeichen): ");
        c = 0;
        while ((cBuffer[c] = getch()) != '\r') /* zeichenweise Einlesen */
            c++;
        cBuffer[c] = '\0';
        if (strlen(cBuffer) > 8)
        {
            printf("\n\nPasswort zu lang. Neueingabe mit <Enter>.");
            printf(" Ende mit <Esc>.");
            cNeu = getche();
        }
        else
        {
            strcpy(cPWort, cBuffer);
            printf("\n\nPasswort gespeichert. Sichtbar machen? (j/n)");
            if ((cZeig = getche()) == 'j')
                printf("\n\nIhr Passwort ist \"%s\"", cPWort);
            cNeu = 27; /* beendet do-while, 'Esc' = 27 */
        }
    } while (cNeu != 27);
    return(0);
}
```

## 9 Zeiger

Die bisher vorgestellten Datentypen ermöglichen es, unabhängig von Speicheradressen zu arbeiten. Es wurde nur mit dem Namen der Variablen gearbeitet und nur das Programm selber wußte, welche Speicheradressen sich hinter den Variablennamen verbergen.

Zu einer Variablen gehören folgende Dinge:

- der Name,
- die Speicheradresse, an der der Wert der Variablen gespeichert ist,
- der Inhalt der Variable, also der Wert an der Speicheradresse und
- der Datentyp, um den Inhalt zu interpretieren und um die Anzahl der Speicherzellen zu bestimmen.

**Beispiel:**

Name	Adresse (hex.)	Inhalt	Datentyp	Interpretation
Zahl1	00003D24h 00003D25h	00000000 00001011	short	11
Zahl2	00003D26h 00003D27h	00000000 00011110	short	30
Zeichen	00003D28h	01000010	char	'B'

Jede Speicheradresse kann 1 Byte = 8 Bits aufnehmen. Die beiden Zahlen im Beispiel sind short-Zahlen (16 Bit-Zahlen) und belegen daher 2 Speicheradressen. Dagegen hat die char-Variable nur 8 Bit und belegt damit auch nur eine Speicheradresse.

### 9.1 Zeiger und Adressen

In C wird an vielen Stellen mit **Zeigern** (im engl. **Pointer**) gearbeitet, um unabhängig von Variablennamen auf die Inhalte der Variablen zugreifen zu können. Ein Zeiger "zeigt" auf eine Speicheradresse, d.h. ein Zeiger ist eine Zahlenvariable und beinhaltet als Wert die Adresse. Dadurch benötigen Zeiger immer den gleichen Speicherplatz (unter Windows 32 Bit, da Windows ein 32-Bit-System ist).

Bei der Deklaration/Definition von Zeigern wird der Datentyp angegeben, auf den der Zeiger zeigen soll. Vor den Variablennamen von Zeigern wird der Präfixoperator \* gesetzt.

**Beispiel:**

```
int *ip;
```

Diese Zeile bewirkt, dass ein Zeiger mit dem Namen ip deklariert und definiert wird. Dieser Zeiger zeigt jetzt grundsätzlich auf Zahlen vom Typ int. Diese Datentypangabe ist wichtig, da mit ihr nicht nur auf die angegebene Speicheradresse, sondern auch noch auf die nächsten 3 zugegriffen wird (weil eine int-Zahl 32 Bit = 4 Bytes = 4 Speicheradressen beinhaltet). Im nächsten Beispiel wird mit einem Zeiger auf eine int-Variable gezeigt.

### Beispiel:

```
int i,*ip; // Deklaration der int-Zahl i und des Zeigers ip
ip = &i;   // Speicheradresse von i in ip speichern
i = 99;
*ip = 100; // weist i die Zahl 100 zu, da ip auf i zeigt
```

Der Adreßoperator & liefert die Speicheradresse eines Objekts. Es wird auch von einer **Referenz** auf das Objekt bzw. vom **Referenzieren** gesprochen. Im Beispiel wird also die Speicheradresse der Variable i dem Zeiger ip zugewiesen. Danach wird der Variable i der Wert 99 zugewiesen. In der letzten Zeile wird dem Inhalt der Speicheradresse, auf die der Zeiger ip zeigt, der Wert 100 zugewiesen. Da der Zeiger auf die Speicheradresse der Variable i zeigt, hat i anschließend den Wert 100. Dies wird durch den Inhaltsoperator \* erreicht. Es wird auch vom **Dereferenzieren** gesprochen.

Hier nochmal eine Tabelle mit verschiedenen Ausdrücken und die im Beispiel resultierenden Werte:

Ausdruck	Ergebnis
i	100
&i	00004711h
ip	00004711h
*ip	100
&ip	00004715h

### Der Zeigerwert NULL

Es gibt noch einen speziellen Zeigerwert, nämlich den Wert NULL. Dieser Zeigerwert zeigt definitiv auf "nichts". Dieser Zeigerwert kann abgefragt werden, z.B.

```
if (ip == NULL) ....
```

Genauso wie die "normalen" Variablen haben Zeiger nach der Definition einen unbekannten Wert, d.h. sie "zeigen" irgendwo hin. Um dies zu vermeiden, sollten Zeiger immer mit dem Zeigerwert NULL initialisiert werden.

## 9.2 Zeiger und Funktionen

Bei der Parameterübergabe an eine Funktion per Zeiger wird der Funktion die Adresse einer Variablen übergeben. Beispiel:

```
int divRound(int x, int y, int *res)
{
    if (y == 0)
        return(-1);
    *res = (float) x / y + 0.5;
    return(0);
}
```

Der Funktion divRound muss beim Aufruf als dritter Parameter die Adresse einer Variablen übergeben werden. Was die Funktion formal erwartet, ist für den dritten Parameter eine



Zeigervariable, was durch `int *res` zum Ausdruck kommt. Es spielt beim Aufruf der Funktion natürlich keine Rolle, ob ihr eine Zeigervariable entsprechend

```
int a, b, ergebnis, *ptr;
ptr = &ergebnis;
/* . . . */
divRound(a, b, ptr);
// übergeben wird oder direkt die Adresse einer Variablen:
int a, b, ergebnis;
/* ... */
divRound(a, b, &ergebnis);
```

## 9.3 Zeiger und Arrays

C interpretiert den Namen eines Feldes als konstanten Zeiger auf das erste Element. In C ist also ein Feld prinzipiell nichts anderes als ein Pointer, der auf den Anfang des Feldes zeigt.

Beispiel:

```
int feld[10], *ptr=NULL;
ptr = &feld[0];    /* ptr = feld; */
```

Mit Zeigern können in C folgende Operationen ausgeführt werden:

1. Zu einem Zeiger kann ein beliebiger ganzzahliger Wert addiert werden.
2. Von einem Zeiger kann ein beliebiger ganzzahliger Wert subtrahiert werden.
3. Subtraktion eines Zeigers von einem anderen: Ergebnis: Anzahl der Elemente, die dazwischenliegen

wichtig: Compiler multipliziert die zu addierende Zahl vorher mit der Größe des Datentyps

Beispiel: Zugriff auf das 5. Element des Feldes `feld`:

```
feld[4]
*(ptr+4)
*(feld+4)
ptr[4]
```

Beispiel:

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

int add_array(int *p, int n);
void denksport(int *zahlen);

int main()
{
    int feld[]={1,5,2,6,4,3}, ergebnis=0;

    ergebnis=add_array(feld,sizeof(feld)/sizeof(int));
    printf("Summe:  %d\n",ergebnis);

    //    (void) denksport(feld);
    return EXIT_SUCCESS;
}
```

```

int add_array(int *p, int n)
{
    int i=0,summe=0;

    for (i=0;i<n;i++)
    {
        summe+=*p;
        p++;
    }
    return summe;
}

void denksport(int *zahlen)
{
    int *ptr=NULL;

    ptr=zahlen+2;
    printf("\n%d",*(ptr+1));
    printf("\n%d",ptr[-1]);
    printf("\n%d",zahlen-ptr);
    printf("\n%d",zahlen[* (ptr++)]);
    printf("\n%d",*(zahlen+zahlen[2]));
    printf("\n%d",*ptr);
    printf("\n");
}

```

## 9.4 Zeigervektoren

Man kann auch Vektoren deklarieren, deren Elemente Zeiger auf einen bestimmten Datentyp sind. Solche Vektoren heißen auch Pointerarrays.

Beispiel:

```
int *zeigervektor[10];
```

..deklariert einen Vektor, dessen Elemente Zeiger auf den Datentyp int sind. Da jeder Zeiger auf einen bestimmten Datentyp in C als Vektor dieses Datentyps interpretiert werden kann (genauer als Anfangsadresse eines solchen Vektors), kann ein Zeigervektor zur Beschreibung eines zweidimensionalen Feldes herangezogen werden.

Beispiel für zweidimensionales Feld vom Typ char:

```

char
*engl_zif[]={ "zero", "one", "two", "three", "four", "five", "six", "seven", "eight",
,nine"}

```

Ausgabe von two: puts(engl\_zif[2]);

## 9.5 Argumentbehandlung

Jedes C-Programm besitzt die Funktion main(), die beim Start des C-Programms von der Betriebssystemebene aus aufgerufen wird. Grundsätzlich werden bei jedem Aufruf von main() 2 Parameter übergeben, die normalerweise mit argc und \*argv[] bezeichnet werden.

**allgemeine Form:**

```

int main(int argc, char *argv[])
{
    ...
    return 0; /* Exit-Code */
}

```

In der zweiten Form werden zwei Parameter übergeben. Über diese beiden Parametern kann auf alle **Kommandozeilenparameter** zugegriffen werden, die beim Aufruf des Programms angegeben wurden. Das folgende Beispiel startet das Programm *prog.exe* mit 4 Kommandozeilenparameter.

### Beispiel:

```
prog 1 Test 547.32 3
```

Der erste Parameter argc ist die Anzahl der Kommandozeilenparameter einschließlich des Programmaufrufs, im Beispiel gleich 5. Der zweite Parameter ist ein String-Array, also ein Array von Zeichenketten. In diesen Zeichenketten stehen die einzelnen Kommandozeilenparameter. Für das obige Beispiel sind folgende Werte in diesem Array gespeichert:

```
argv[0] = "prog"
argv[1] = "1"
argv[2] = "Test"
argv[3] = "547.32"
argv[4] = "3"
argv[5] = "" (bzw. = '\0', Stringende)
```

**Wichtig:** Auch die Zahlen-Kommandozeilenparameter werden hier als Texte gespeichert.

### Beispiel:

```
#include <stdio.h>

int main(int argc, char* argv[])
{
    int i = 1;
    printf("\n\nProgrammaufruf: %s\n", argv[0]);
    if (argc == 1)
    {
        printf("Keine Kommandozeilenparameter\n");
    }
    else
    {
        printf("Anzahl Kommandozeilenparameter: %i\n", argc - 1);
        printf("Kommandozeilenparameter:\n");
        while (argv[i])
        {
            printf("%2d: %s\n", i, argv[i]);
            i++;
        }
    }
    return 0;
}
```

Funktionen, die in Zukunft verwendet werden sollen:

```
void Usage(void)
{
    (void) fprintf(stderr, "Usage: %s a n\n", szCommand);
    BailOut((const char *)0);
}

void BailOut(const char *szMessage)
{
    if (szMessage != (const char *)0)          /* print error message */
    {
        (void) fprintf(stderr, "%s: %s\n", szCommand, szMessage);
    }
    exit(EXIT_FAILURE);
}
```

# 10 Strukturen und Aufzählungstypen

## 10.1 Strukturen

Bei Arrays werden viele Elemente desselben Datentyps zusammengefasst. Häufig ist aber gewünscht, auch Daten unterschiedlichen Typs zu einem Objekt zusammenzufassen, wenn diese logisch zusammengehören. Die Lösung dazu hat in C den Namen struct und ist folgendermaßen definiert:

```
struct [Typname]
{   Aufbau
} [Variablenliste];
```

Man kann sich dies als Karteikarte vorstellen. Um zum Beispiel eine Bücherverwaltung für die Bibliothek zu erstellen, müssen die Daten eines Buches - Titel, Autor, Standort, usw. - als ein Objekt behandelt werden. Diese einzelnen Daten werden im Aufbau wie gewöhnliche Variablen angegeben. Im folgenden Beispiel wird mit Hilfe von struct ein Datentyp namens Buch (Typname) definiert; gleichzeitig wird ein Array dieses Datentyps mit dem Namen Buecher erstellt (Variablenliste).

Beispiel:

```
struct Buch
{   char Titel[100];
    char Autor[100];
    char ISBN[20];
    char Standort[10];
    float Preis;
} Buecher[50];      // 50mal struct Buch
```

Wahlweise kann der Typname oder die Variablenliste weggelassen werden. Wird der Typname weggelassen, werden Variablen des strukturierten Datentyps definiert, aber dieser strukturierte Datentyp hat keinen Namen. Dadurch können später im Programm keine weiteren Variablen dieses Datentyps definiert werden, es sei denn, es wird wieder die komplette Typdefinition geschrieben.

Wird anders herum die Variablenliste weggelassen, wird ein strukturierter Datentyp mit Namen definiert, es werden aber keine Variablen dieses Typ definiert. Dies kann aber später im Programm noch erfolgen:

```
struct Buch Buch1;
```

Auf die Elemente eines strukturierten Datentyps kann nicht direkt zugegriffen werden, weil sie nur in Verbindung mit dem Objekt existieren. Die Anweisung Preis = 19.99; ist unsinnig, weil nicht klar ist, welches Buch diesen Preis hat. Der Zugriff auf die Elemente einer Variablen vom Typ struct geschieht über einen Punkt zwischen Variablen- und Elementnamen.

z.B Buch1.Preis = 19.99;

Dann gilt der Preis für das Buch Buch1.

### Beispiel:

```
/* **** */
/* Dieses Programm legt zwei Variablen */
/* eines strukturierten Datentyps an.   */
/* **** */
#include <stdio.h>
#include <string.h>

void main()
{
    struct Buch
    {
        char Titel[100];
        char Autor[100];
        char Standort[10];
        double Preis;
    } Buch1;    // direkt bei der Definition des structs

    struct Buch Buch2; // Definition nur über Typnamen

    strcpy(Buch1.Titel, "Das indische Tuch");
    strcpy(Buch1.Autor, "Edgar Wallace");
    strcpy(Buch1.Standort, "Reihe 5");
    Buch1.Preis=14.90;
    Buch2.Preis=3.5;
}
```

Andere Möglichkeit ein struct zu definieren:

Das Schlüsselwort `typedef` erlaubt die Zuweisung eines Namens an beliebige Datentypen.

### Beispiel:

```
typedef struct
{
    char Titel[100];
    char Autor[100];
    char Standort[10];
    double Preis;
} Buch;    // Definition eines neuen Datentyps Buch

Buch Buch1, Buch2; // Definition zweier Objekte der obigen Struktur
```

## 10.2 Zeiger und Strukturen

### Beispiel:

```
struct Person
{
    int Personalnummer;
    char Name[50];
    char Anschrift[100];
    float Gehalt;
};

struct Person Otto, *zeiger=NULL;
zeiger = &Otto;
```

Auf die Komponenten von Otto kann entweder direkt mit z.B. `Otto.Personalnummer` oder indirekt über den Zeiger zugegriffen werden. Der Zugriff über den Zeiger würde lauten:

```
(*zeiger).Personalnummer
```

oder in der vorwiegend benutzten Form:

```
zeiger->Personalnummer
```

## 10.3 Aufzählungstypen

Häufig gibt es nicht-numerische Wertebereiche. So kann beispielsweise ein Wochentag nur die Werte Sonntag, Montag, ..., Samstag annehmen. Eine mögliche Hilfskonstruktion ist das Verwenden von ganzen Zahlen (int), wobei jeweils eine Zahl einem nicht-numerischen Wert entspricht (z.B. 0 = Sonntag, 1 = Montag, usw.). Dabei muss die Bedeutung der einzelnen Zahlen irgendwo als Kommentar festgehalten werden. Dadurch ist das Programm wieder deutlich schlechter lesbar.

Die Lösung für solche Fälle sind Aufzählungstypen. Die Syntax dazu sieht so aus:

```
enum [Typname] {Aufzählung} [Variablenliste];
```

Ähnlich wie bei struct kann hier wahlweise der Typname oder die Variablenliste weggelassen werden.

Beispiel:

```
enum Wochentag {Sonntag, Montag, Dienstag, Mittwoch,  
                Donnerstag, Freitag, Samstag};
```

Damit ist der Aufzählungstyp definiert. Nun lassen sich Variablen dieses Typs definieren:

```
enum Wochentag Werktag, Feiertag, Heute = Dienstag;
```

Diesen Variablen können nur Werte aus der Werteliste des Aufzählungstypen zugewiesen werden. Intern werden sie auf die ganzen Zahlen - beginnend bei 0 - abgebildet. Dadurch ist eine implizite Typumwandlung von einem Aufzählungstypen zu den ganzen Zahlen möglich, aber nicht umgekehrt!

```
Werktag = Montag; //richtig  
Feiertag = Sonntag; //richtig  
int i = Freitag; //richtig (implizite Typumwandlung)  
Heute = 4; //falsch! (keine Typumwandlung möglich!)  
int i = Montag + Dienstag; // richtig (aber sinnlos)  
Heute = Montag + Dienstag; // falsch!  
Heute++; // falsch!
```

Die Werte der Werteliste von Aufzählungstypen sind Konstanten und können nach der Definition nicht mehr verändert werden. Wohl aber können bei der Definition des Aufzählungstypen den Werten andere Zahlen zugewiesen werden. Dazu wird jedem Wert gleich die zugehörige Zahl zugewiesen. Jede Zahl darf dabei nur einmal verwendet werden.

Beispiel:

```
enum Farben {weiss = 0, blau = 2, gruen = 5, rot = 25,  
            gelb = 65, lila = 90, pink = 99};
```

## 11 Dynamische Speicherverwaltung

Eine Personalverwaltungssoftware in einem Unternehmen soll immer dann, wenn ein neuer Mitarbeiter in ein Unternehmen eintritt, die entsprechenden Personaldaten speichern. Eine Möglichkeit, den entsprechenden Speicherplatz zur Verfügung zu stellen, ist, ein Feld zu definieren, in dem die Feldelemente Strukturen zur Aufnahme von Personaldaten sind.

```
struct Person
{
    int Personalnummer;
    char Name[50];
    /* und weitere */
};

struct Person Mitarbeiter[200];
```

Dieser Ansatz hat den Nachteil, dass er recht unflexibel ist. Die maximal mögliche Anzahl von Beschäftigten, die mit dieser Personalverwaltungssoftware verwaltet werden können, muss in der Entwicklungsphase der Software festgelegt werden (im Beispiel hat das Feld Mitarbeiter maximal Platz zum Speichern von 200 Personen). Was passiert, wenn das Unternehmen wächst und irgendwann doch mehr als die maximal geplanten 200 Personen beschäftigt werden sollen. Eine mögliche Lösung wäre, ein Feld mit vielleicht 5000 Feldelementen zu definieren, womit man sicherlich bei einem geplanten Maximum von 200 Personen auf der sicheren Seite wäre.

Diese Art der Lösung ist ungeschickt und dem Problem nicht gerade angepasst. Eine enorme Speicherverschwendung ist die Folge. Geschickter wäre ein Ansatz, bei dem Speicher erst dann zur Verfügung gestellt wird, wenn er auch benötigt wird. Das heißt konkret, dass der zur Speicherung von bestimmten Daten benötigte Speicherplatz erst zur Laufzeit des Programms und nicht schon vorher verfügbar gemacht wird. Für unser Beispiel würde dies bedeuten, dass immer dann, wenn ein neuer Mitarbeiter ins Unternehmen eintritt, eine neue Variable vom Typ `struct Person` erzeugt wird. Sollte umgekehrt ein Mitarbeiter das Unternehmen verlassen, werden nicht nur seine Daten gelöscht, sondern auch die entsprechende Variable wieder vernichtet.

Zur Realisierung dieses Konzeptes bietet C im Wesentlichen die Standardfunktionen `malloc` und `free` an. Mit der Funktion `malloc` können Variablen zur Laufzeit des Programms (dynamisch) erzeugt, mit der Funktion `free` wieder vernichtet werden.

```
void *malloc(size_t size)
void free(void *zeiger)
```

`size_t` ist ein selbstdefinierter Datentyp und bei Microsoft identisch mit `unsigned integer`. Der Funktion `malloc` wird die Anzahl der Bytes übergeben, die für die zu erzeugende Variable benötigt werden. Die Funktion `malloc` allokiert diese Anzahl von Bytes im Speicher und gibt die Anfangsadresse dieses neuen Speicherbereichs als Rückgabewert in Form eines Zeigers zurück.

Soll die mit `malloc` erzeugte Variable wieder vernichtet werden, wird die Funktion `free` aufgerufen und ihr der entsprechende Zeiger als Parameter übergeben. Die Funktion `free` gibt daraufhin den zuvor allokierten Speicherbereich wieder frei.

Beispiel: Erzeugung einer Variablen vom Typ `int`

```
int *ptrInt;
ptrInt = (int *) malloc(sizeof(int));
free(ptrInt);
```



Die Anzahl der zu allozierenden Bytes wird nicht direkt als Zahl, sondern mittels des `sizeof`-Operators übergeben (eine Variable vom Typ `int` kann ja in einer Umgebung 2 Byte groß sein, in anderen aber 4 Bytes). Der von `malloc` zurückgegebene Zeiger muss vor der Zuweisung an `ptrInt` noch gecastet werden (`int *`). Soll die Variable, auf die `ptrInt` zeigt, wieder vernichtet werden, wird `ptrInt` einfach der Funktion `free` als Parameter übergeben. Genauso wie eine Variable vom Typ `int` lässt sich natürlich auch eine Variable von jedem anderen Typ erzeugen, zum Beispiel vom Typ `struct Person`:

```
struct Person *ptrPerson;
ptrPerson = (struct Person *) malloc(sizeof(struct Person));
free(ptrPerson);
```

Es lassen sich auch Felder von Variablen dynamisch erzeugen: `void *calloc( size_t num, size_t size );`

```
int *ptrInt;
ptrInt = (int *) calloc(12, sizeof(int));
free(ptrInt);
```

`ptrInt` zeigt in diesem Fall nicht auf eine einzelne Variable, sondern auf ein Feld von 12 Variablen vom Typ `int`. Der Zugriff auf die einzelnen Feldelemente geschieht einfach durch z.B: `ptrInt[3]` oder `*(ptrInt+4)`

Sollte zur Erzeugung einer neuen Variablen kein Speicherplatz mehr frei sein, gibt die Funktion `malloc` den NULL-Pointer `NULL` zurück. Es ist daher vernünftig, immer abzufragen, ob genug Speicherplatz verfügbar war:

```
if ((ptrInt = (int *) malloc(sizeof(int))) == NULL)
{
    printf("\nKein Speicher mehr verfügbar.");
    /* eventuell noch exit(1), oder BailOut(...) */
}
else
{
    /* weiter geht es im Programm */
}
```

Ein böser Fehler, der zu unvorhergesehenem Programmverhalten führt (unpredictable result) und daher nur schwer zu lokalisieren ist, ist, die Funktion `free` aufzurufen, ohne vorher mittels `malloc` Speicher allokiert zu haben. Dieser Fehler kann z.B. auftreten, wenn nicht abgefragt wurde, ob der Rückgabewert von `malloc` `NULL` war. Ein derartiger Fehler kann aber auch passieren, wenn in einer komplexeren Programmstruktur die Funktion `free` in verschiedenen Programmzweigen aufgerufen wird, wobei man sich sicher ist, dass das Programm niemals mehr als einen Programmzweig durchläuft. Dumm gelaufen, wenn das Programm doch mehr als einen Programmzweig durchläuft.

Eine gute Verteidigungsstrategie gegen diese Art von Fehlern ist, jedes mal vor dem Aufruf der Funktion `free` zu überprüfen, ob der entsprechende Zeiger einen Wert ungleich `NULL` hat. Wenn ja, wird `free` ausgeführt und anschließend der Zeiger explizit auf `NULL` gesetzt.

Beispiel:

```
if (ptrInt != NULL)
{
    free(ptrInt);
    ptrInt = NULL;
}
```

## 12 Dateibehandlung in C

### 12.1 Allgemeines

In C operieren alle Funktionen für die Datenein- und Ausgabe auf sogenannten Streams. Ein Stream ist eine sequentielle Folge von Zeichen, wobei dieser mit einem Ein- oder Ausgabegerät oder auch mit einem File verbunden sein kann. Auf jedem Stream kann man entweder sequentiell Zeichen schreiben oder sequentiell Zeichen lesen. Auf Streams, die mit Files verbunden sind, kann außerdem der Schreib-Lesezeiger an einer anderen Stelle im File positioniert werden.

Es existieren grundsätzlich 2 verschiedene Arten von Streams:

- Text-Streams und
- binäre Streams

Bei Lese- und Schreiboperationen ist zwischen ungepufferten und gepufferten zu unterscheiden. Im ersten Fall wird immer direkt auf die Datei zugegriffen, im zweiten Fall greift das Programm auf einen Puffer zu, der vom Betriebssystem verwaltet wird. In den Puffer werden einzulesende Daten "auf Vorrat" geholt bzw. auszugebende Daten gesammelt, um die Anzahl zeitaufwendiger Zugriffe auf externe Speichermedien zu reduzieren.

Gepufferte Ausgabe kann zu Problemen führen:

Das Programm nimmt an, die von ihm ausgegebenen Daten seien im externen Medium gespeichert. Das externe Medium enthält die ausgegebenen Daten nicht vollständig, da sich noch Teile der zuletzt geschriebenen Daten im Puffer befinden.

3 Streams sind beim Start jedes Programmes verfügbar:

- stdin
- stdout
- stderr

### 12.2 Eine Datei öffnen

Vor der Verwendung von `fopen` muss ein Stream, d.h ein Zeiger auf den Typ `FILE` erzeugt werden:

```
FILE *fp;
```

Dieser Stream wird nun geöffnet, wobei überprüft wird, ob dabei ein Fehler aufgetreten ist. Scheitert der Aufruf von `fopen()`, lautet der Rückgabewert `NULL`. Ein solches Scheitern kann durch einen Hardware-Fehler ausgelöst werden oder durch einen Dateinamen, dessen Pfadangaben ungültig sind.

#### Prototyp von `fopen()`:

```
FILE *fopen(const char *filename, const char *mode);
```

Das Argument `filename` ist der Name der Datei, die geöffnet werden soll.

Das Argument `mode` gibt den Modus an, in dem die Datei geöffnet werden soll.

Zulässige Werte:

Modus	Bedeutung
r	Öffnet die Datei zum Lesen. Wenn die Datei nicht existiert, liefert fopen() NULL zurück.
w	Öffnet die Datei zum Schreiben. Wenn es noch keine Datei des angegebenen Namens gibt, wird sie erzeugt. Existiert bereits eine Datei mit gleichlautendem Namen, wird sie ohne Warnung gelöscht und eine neue, leere Datei erzeugt.
a	Öffnet die Datei zum Anhängen. Wenn es noch keine Datei des angegebenen Namens gibt, wird sie erzeugt. Existiert die Datei, werden die neuen Daten an das Ende der Datei angehängt.
r+	Öffnet die Datei zum Lesen und Schreiben. Wenn die Datei nicht existiert, liefert fopen() NULL zurück
w+	Öffnet die Datei zum Lesen und Schreiben. Wenn es noch keine Datei des angegebenen Namens gibt, wird sie erzeugt. Existiert die Datei, wird sie überschrieben.
a+	Öffnet die Datei zum Lesen und Anhängen. Wenn es noch keine Datei des angegebenen Namens gibt, wird sie erzeugt. Existiert die Datei, werden die neuen Daten an das Ende der Datei angehängt.
b	Bearbeitung im Binär-Modus
t	Bearbeitung im Text-Modus (default)

## 12.3 Schließen der Datei

**Prototyp:**

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

schließt ein File. Zum Schreiben gepufferte Daten werden geschrieben, zum Lesen gepufferte Daten werden verworfen, der Puffer wird freigegeben und der Stream vom File getrennt. Nach einem fclose() ist der Stream nicht mehr verwendbar. Rückgabewert: 0 bei Erfolg

## 12.4 Schreiben und Lesen

Daten können auf drei Arten in eine Datei geschrieben oder von der Datei gelesen werden:

Bei der **formatierten Ausgabe** schreibt man formatierte Textdaten in eine Datei.

Bei der **Zeichenausgabe** schreibt man einzelne Zeichen oder ganze Zeilen in eine Datei.

Bei der **direkten Ausgabe** schreibt man den Inhalt eines Speicherabschnitts direkt in eine Datei. Diese Methode wird nur bei binären Dateien angewendet.

### 12.4.1 Formatierte Dateiausgabe

Für formatierte Dateiausgaben verwendet man die Bibliotheksfunktion `fprintf()`. Der Prototyp von `fprintf()` steht in der Header-Datei `stdio.h` und lautet:

```
int fprintf(FILE *fp, const char *format, ...);
```

Das erste Argument ist ein Zeiger auf den Typ `FILE` (Zeiger, der von `fopen()` zurückgeliefert wurde).

Das zweite Argument ist der Formatstring – siehe Funktion `printf()`.

### 12.4.2 Formatierte Dateieingabe

Für die formatierte Dateieingabe steht die Bibliotheksfunktion `fscanf()` zur Verfügung, die bis auf die Ausnahme, dass die Eingabe von einem angegebenen Stream anstatt von `stdin` kommt, der Funktion `scanf()` entspricht.

#### Prototyp:

```
int fscanf(FILE *fp, const char *format, ...);
```

Wird ein unerwartetes Zeichen von `fp` gelesen, so wird es in den Stream zurückgestellt und `fscanf` bricht ab.

Achtung: `fscanf` kann, wenn bei String-Leseoperationen keine Feldweite angegeben wird, über das Ende des bereitgestellten Arrays hinausschreiben.

Aus diesem Grund ist es besser, statt `fscanf` die Funktion `fgets`, gefolgt von `sscanf` zu verwenden. (Prototyp: `int sscanf (const char *s, const char *format, ...);` entspr. `fscanf`, liest allerdings von einem String `s`).

### 12.4.3 Zeicheneingabe und -ausgabe

Im Zusammenhang mit Dateien versteht man unter dem Begriff Zeichen-E/A das Einlesen sowohl einzelner Zeichen als auch ganzer Zeilen. Eine Zeile ist eine Folge von null oder mehr Zeichen, die mit einem Neue-Zeile-Zeichen (CR-LF) abschließen.

### 12.4.4 Zeicheneingabe

Die Funktionen `getc()` und `fgetc()` sind identisch und damit austauschbar. Sie lesen ein Zeichen aus einem angegebenen Stream ein.

#### Prototyp:

```
int getc(FILE *fp);  
int fgetc(FILE *fp);
```

Das Argument `fp` ist der Zeiger, der nach Öffnen der Datei von `fopen()` zurückgeliefert wurde. Die Funktion liefert das eingegebene Zeichen zurück oder **EOF**, wenn ein Fehler aufgetreten ist.

Um eine ganze Zeile von Zeichen aus einer Datei einzulesen, verwendet man die Bibliotheksfunktion `fgets()`.

**Prototyp:**

```
char *fgets(char *str, int n, FILE *fp);
```

Das Argument `str` ist ein Zeiger auf einen Puffer, in dem die Eingabe gespeichert wird. `n` ist die maximale Anzahl der einzulesenden Zeichen. `fp` ist der Zeiger auf den Typ `FILE`, der beim Öffnen der Datei von `fopen()` zurückgegeben wurde.

Wenn `fgets()` aufgerufen wird, liest die Funktion Zeichen aus `fp` ein und schreibt sie an die Stelle im Speicher, auf die `str` zeigt. Es werden so lange Zeichen gelesen, bis ein Neue-Zeile-Zeichen erscheint oder `n-1` Zeichen eingelesen wurden - je nachdem welcher Fall zuerst eintritt. Man setzt `n` gleich der Anzahl der Bytes, die für den Puffer `str` reserviert wurden und damit wird verhindert, dass Speicher über den reservierten Bereich hinaus beschrieben wird. (Mit `n-1` stellt man sicher, dass Platz für das abschließende Nullzeichen `\0` gelassen wird, das `fgets()` an das Ende des Strings anhängt). Im fehlerfreien Fall retourniert die Funktion `s` (falls mindestens 1 Zeichen gelesen wurde). Wenn das Ende der Datei erreicht wurde und keine Zeichen in das Array gelesen wurden, oder wenn ein Lesefehler passiert ist, liefert die Funktion **NULL** zurück.

## 12.4.5 Zeichenausgabe

**Prototyp:**

```
int putc(int c, FILE *fp);  
int fputc(int c, FILE *fp);
```

Sie schreibt das Zeichen `c` (in `unsigned char` umgewandelt) auf `fp`. Tritt dabei ein Fehler auf, so wird EOF zurückgeliefert, sonst `c`.

**Prototyp:**

```
int fputs(char *str, FILE *fp);
```

Das Argument `str` ist ein Zeiger auf den auszugebenden nullterminierten String und `fp` ein Zeiger auf den Typ `FILE`, der beim Öffnen der Datei von `fopen()` zurückgeliefert wurde. Der String, auf den `str` zeigt, wird ohne das abschließende `\0` in die Datei geschrieben. Die Funktion `fputs()` liefert im Erfolgsfall die Anzahl der übertragenen Zeichen zurück oder EOF bei einem Fehler.

```

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

int main(void)
{
    FILE *fp;
    char zeile[100];
    char text[]="3. Datensatz\n";
    int i=0;

    //Neue Datei erzeugen
    if ((fp=fopen("test.dat", "wt"))==NULL)
    {
        fprintf(stderr, "Fehler beim Öffnen");
    }
    else
    {
        fprintf(fp, "1. Datensatz\n");
        fprintf(fp, "2. Datensatz\n");
    }
    if (fclose(fp)) fprintf(stderr, "Fehler");

    /*Datei zum Lesen und Schreiben öffnen (Datei wird dabei aber nicht
    gelöscht) */
    if ((fp=fopen("test.dat", "r+t"))==NULL)
    {
        fprintf(stderr, "Fehler beim Öffnen");
    }
    else
    {
        // fseek(fp, sizeof(text), SEEK_SET);
        fprintf(fp, "3. Datensatz\n");
    }
    if (fclose(fp)) fprintf(stderr, "Fehler");

    //Datei zum Lesen öffnen
    if ((fp=fopen("test.dat", "rt"))==NULL)
    {
        fprintf(stderr, "Fehler beim Öffnen");
    }
    else
    {
        //Zeilenweise den Inhalt der Datei auslesen
        while((fgets(zeile, sizeof(zeile)-1, fp)!=NULL))
        {
            printf("%s", zeile);
        }
    }
    if (fclose(fp)) fprintf(stderr, "Fehler");

    //Datei "test.dat" wieder löschen
    if (remove("test.dat")==-1){
        fprintf(stderr, "Fehler beim Loeschen der Datei");
    }
    return EXIT_SUCCESS;
}

```

## 12.4.6 Direkte Dateieingabe und -ausgabe

Befehle wie `fprintf` oder `fscanf` bewirken, dass die Werte auf dem externen Medium als ASCII-Zeichen gespeichert sind. -> Umwandlung benötigt Zeit.

Die direkte Datei-E/A ist nur bei binären Dateien sinnvoll. Bei der direkten Ausgabe werden Datenblöcke vom Speicher in eine Datei geschrieben. Die direkte Dateieingabe kehrt diesen Prozess um: Ein Datenblock wird von einer Datei in den Speicher gelesen.

Die Funktion `fwrite()`

### Prototyp:

```
size_t fwrite(void *ptr, size_t size, size_t nmemb, FILE *fp);
```

Sie schreibt `nmemb` Elemente von jeweils `size` Bytes Größe aus dem Array, auf das `ptr` zeigt, auf den Stream `fp`. Die Anzahl der erfolgreich geschriebenen Elemente wird zurückgeliefert (welche im Fehlerfall kleiner als `nmemb` ist).

Will man auf Fehler prüfen, ruft man `fwrite()` in der Regel wie folgt auf:

```
if( (fwrite(puffer, groesse, anzahl, fp)) != anzahl) fprintf(stderr,
"Fehler beim Schreiben in die Datei.");
```

Beispiele: `x` vom Typ `double` in eine Datei schreiben:

```
fwrite(&x, sizeof(double), 1, fp);
```

Um ein Array `daten[]` mit 50 Strukturen vom Typ `adresse` in eine Datei zu schreiben, haben Sie zwei Möglichkeiten:

```
fwrite(daten, sizeof(adresse), 50, fp);
fwrite(daten, sizeof(daten), 1, fp);
```

Im ersten Beispiel wird das Array als Folge von 50 Elementen ausgegeben, wobei jedes Element die Größe einer Struktur vom Typ `adresse` hat. Das zweite Beispiel behandelt das Array als ein einziges Element. Das Resultat ist für beide Aufrufe das Gleiche.

Die Funktion `fread()`

Die Bibliotheksfunktion `fread()` liest einen Datenblock aus einer Datei in den Speicher. `fread()` wird normalerweise zum Lesen von binären Daten verwendet.

### Prototyp:

```
size_t fread(void *ptr, size_t size, size_t nmemb, FILE *fp);
```

Sie liest `nmemb` Elemente von jeweils `size` Bytes Größe von `fp` und schreibt sie in das Array, auf das `ptr` zeigt. Die Anzahl der erfolgreich gelesenen Elemente wird zurückgeliefert. Dieser Wert kann kleiner als `nmemb` sein, wenn vorher das Dateiende erreicht wurde oder ein Fehler aufgetreten ist.

## Die Funktion `fflush`

Der Puffer eines Streams kann mit der Bibliotheksfunktion `fflush()` geleert werden, ohne ihn zu schließen. Sie sendet alle Daten, die sich im Ausgabepuffer des Streams befinden, an das Betriebssystem. Ist `fp` ein `NULL`-Pointer, so werden alle Ausgabestreams geflushet.

### Prototyp:

```
int fflush(FILE *fp);
```

Im Erfolgsfall liefert `fflush()` 0 und bei Auftreten eines Fehlers `EOF` zurück.

Um das Ende einer Binärdatei bestimmen zu können, verwendet man die Funktion `feof()`.

### Prototyp:

```
int feof(FILE *fp);
```

Sie liefert einen Wert ungleich Null, wenn das Dateiende erreicht ist.

## 12.4.7 Direktzugriff

Bis jetzt wurde die Position des Datenzeigers, der innerhalb der Datei auf die jeweilige Position des Schreib- bzw. Lesevorganges zeigt, nicht direkt verändert (sequentieller Zugriff auf eine Datei). Sogenannte Positionierfunktionen dienen dazu, den Schreib-Lese-Zeiger eines Streams auf eine andere Stelle zu positionieren.

Um den Positionszeiger auf den Anfang der Datei zu setzen, verwendet man die Bibliotheksfunktion `rewind()`.

### Prototyp:

```
void rewind(FILE *fp);
```

Nach dem Aufruf von `rewind()` wird der Positionszeiger der Datei auf den Anfang der Datei (Byte 0) gesetzt.

Um den Wert des Positionszeigers einer Datei zu ermitteln, verwendet man `ftell()`.

### Prototyp:

```
long ftell(FILE *fp);
```

Die Funktion `ftell()` liefert einen Wert vom Typ `long` zurück, der die aktuelle Dateiposition als Abstand in Byte vom Beginn der Datei angibt (das erste Byte steht an Position 0). Wenn ein Fehler auftritt, liefert `ftell()` `-1L` (der Wert -1 des Typs `long`).

Mehr Kontrolle über den Positionszeiger eines Streams bietet die Bibliotheksfunktion `fseek()`. Mit dieser Funktion kann der Positionszeiger an eine beliebige Stelle in der Datei versetzt werden.



**Prototyp:**

```
int fseek(FILE *fp, long offset, int ausgangspunkt);
```

Die Distanz, um die der Positionszeiger verschoben wird, wird in `offset` in Byte angegeben. Das Argument `ausgangspunkt` gibt die Position an, von der aus die Verschiebung berechnet wird. Der Parameter `ausgangspunkt` kann drei verschiedene Werte annehmen, für die in `stdio.h` symbolische Konstanten definiert sind (siehe Tabelle 15.2).

Konstante	Wert	Beschreibung
SEEK_SET	0	Verschiebt den Anzeiger um <code>offset</code> Byte vom Beginn der Datei
SEEK_CUR	1	Verschiebt den Anzeiger um <code>offset</code> Byte von der aktuellen Position
SEEK_END	2	Verschiebt den Anzeiger um <code>offset</code> Byte vom Ende der Datei

Die Funktion `fseek()` liefert 0 zurück, wenn der Positionszeiger erfolgreich verschoben wurde, beziehungsweise einen Wert ungleich Null, wenn ein Fehler aufgetreten ist.

!!!!!!!!!!!!!!!!!!!!!! Beispielprogramm ADIM Schülerkartei !!!!!!!!!!!!!!!!!!!!!!!

# 13 Prüfen von Programmen

Fast jedes neue Programm enthält zunächst einige Programmierfehler. Selbstverständlich müssen diese Fehler möglichst vollzählig entfernt werden, bevor man das Programm verwenden kann. Es ist jedoch nicht leicht, alle Fehler zu finden.

## 13.1 Arten von Fehlern

### 13.1.1 Syntaktische Fehler

Syntaktische Fehler treten in einem Programm dort auf, wo die Regeln der Programmiersprache verletzt worden sind.

Beispiele:

- Name nicht definiert (Variable, Konstante, Funktion nicht vereinbart)
- ";" ", " oder anderes Trennzeichen vergessen (zwischen den Anweisungen oder zwischen den aktuellen Parametern eines Funktionsaufrufs)
- Anzahl der öffnenden und schließenden Klammern bei einem arithmetischen Ausdruck oder Start und Ende von Blöcken stimmen nicht überein
- Falsche Parameteranzahl bzw falsche Parametertypen bei Funktionsaufrufen

Syntaktische Fehler werden immer vom Compiler erkannt. Gute Compiler geben auch ziemlich genaue Meldungen, weniger komfortable oft nur sehr sparsame wie etwa "syntax error". Manche syntaktische Fehler werden erst später erkannt (zB Fehlendes Blockende erst beim Beginn der folgenden Funktion).

### 13.1.2 Laufzeitfehler

Ist ein Programm frei von syntaktischen Fehlern, so kann es vom Compiler übersetzt werden. Während des Programmablaufs können weitere Fehler auftauchen und zum Programmabbruch führen. Diese Fehler nennt man Laufzeitfehler.

Beispiele:

- Division durch 0
- Speicherzugriffsverletzung
- Fehlendes Datenfile

Es ist möglich, dass der Laufzeitfehler erst lange nach der Stelle auftritt, an der das Programm fehlerhaft ist.

### 13.1.3 Logische Fehler

Wenn ein Programm bei den Probeläufen zum normalen Ende kommt ("nicht abstürzt") und Ergebnisse liefert, so können diese Ergebnisse noch falsch sein: Der Programmierer hat dann einen logischen Fehler gemacht. Man erhält in diesem Fall keinen Hinweis mehr darauf, dass das Programm falsch ist. Man muss vielmehr selbst die Ergebnisse kontrollieren und, falls sie sich als falsch erweisen, die Programmierfehler finden und korrigieren.

## 13.2 Testen

Das Entfernen von Programmierfehlern geht in zwei Phasen vor sich, die sich allerdings oft wiederholen können: das Feststellen von Fehlern und das Korrigieren der festgestellten Fehler.

### 13.2.1 Feststellen von Fehlern

In der ersten Phase versucht man nur, möglichst viele Fehler zu finden. Mit dem Testen eines Programms beginnt man schon sehr früh. Von Zeit zu Zeit prüft man Abschnitte des Programms "mit der Hand", d.h. der Programmierer selbst "spielt Computer". Dabei sollte man folgende Regeln beachten:

- Jede Anweisung exakt ausführen – wenn man sich darauf verlässt, dass die Anweisungen "schon das Richtige tun", übersieht man unweigerlich Fehler.
- Übersichtliche Datensätze wählen
- Kritische Daten wählen – das sind Daten bei denen Sonderfälle auftreten oder bei denen Fehler besonders leicht sichtbar sind.

Erst wenn das manuelle Testen vermuten lässt, dass das Programm korrekt ist, und wenn alle syntaktischen Fehler entfernt worden sind, kann man auch das übersetzte Programm testen.

Außerdem muss man beim Testen eines Programms darauf achten, dass alle Zweige aller IF-Anweisungen mindestens einmal durchlaufen werden. Wenn ein Programm Schleifen enthält, soll man für jede Schleife Testdaten vorsehen, die bewirken, dass die Schleife nullmal, einmal und "oft" durchlaufen wird.

Bei großen Programmen sind oft weitere Maßnahmen notwendig, zB Unterteilung des Programms und getrenntes Prüfen der einzelnen Teile. Das Testen von Programmen sollte nicht nur durch den Programmierer selbst erfolgen, da dieser oft "programmblind" ist und nur Testdaten verwendet, die "sein" Programm nicht abstürzen lassen.

### 13.2.2 Korrigieren der Fehler

Wenn man durch Testläufe Fehler in einem Programm festgestellt hat, muss man diese Fehler auch lokalisieren und korrigieren. Dazu ist es notwendig, dass man genau weiß, was das Programm mit Hilfe welcher Anweisungen bewirken soll. Kann man den Fehlerort trotzdem nicht lokalisieren, muss man Zwischenergebnisse ausgeben. An verschiedenen Stellen fügt man Ausgabeanweisungen für charakteristische Zwischenergebnisse ein. So kann man auf fehlerhafte Abschnitte schließen und dort die Fehler suchen. Folgende Regeln sollte man beachten:

- Lieber zu viele als zu wenige Daten ausgeben
- Sofort nach der Eingabe der Daten die Daten wieder ausgeben. So kann man sicherstellen, dass die Eingabe richtig programmiert wurde und dass die Daten richtig eingegeben wurden.
- Alle ausgegebenen Testdaten auf Korrektheit überprüfen.

Wenn man einen Fehler gefunden und ausgebessert hat, muss man das Programm wieder mit allen Testdaten testen. Durch die Korrektur können sich nämlich neue Fehler eingeschlichen haben, die sich bei anderen Eingabedaten auswirken.

Merke:

- Liefert ein Programm falsche Ergebnisse, dann ist es falsch
- Liefert ein Programm richtige Ergebnisse, dann muss es nicht richtig sein. Bei weiteren Durchläufen mit anderen Daten können völlig falsche Ergebnisse erzeugt werden oder Laufzeitfehler auftreten.

Die sorgfältige Prüfung eines Programms erfordert meistens viel mehr Zeit und Mühe als das Programmieren.

# 14 Such- und Sortialgorithmen

Datensätze bestehen aus verschiedenen Teilen (vgl. Strukturen). Ein Teil speichert den Schlüssel des Datensatzes, über den die Identifikation des Datensatzes erfolgt. Die weiteren Teile beinhalten Zusatzinformationen.

Beispiel:

Personendaten (Name, Adresse, Telefonnummer)  
Schlüssel: Name

Die Anordnung der Datensätze kann in unterschiedlichen Datenstrukturen erfolgen.

Beispiele für Datenstrukturen

- **Feld**  
Zugriff auf ein Element über einen Index
- **Liste**  
Element besitzt Verweis auf das folgende Element
  - **Stapel (Stack)**  
Elemente können nur in umgekehrter Reihenfolge des Einfügens gelesen oder gelöscht werden
  - **Warteschlange (Queue)**  
Elemente können nur in gleicher Reihenfolge des Einfügens gelesen oder gelöscht werden
- **Graphen, Bäume**  
Elemente besitzen variable Anzahl von Verweisen auf weitere Elemente

## 14.1 Laufzeitkomplexität von Algorithmen

**Beispiele für Suchalgorithmen:**

- lineare/sequentielle Suche
- binäre Suche

**Beispiele für Sortierv Verfahren**

- Bubble-Sort
- Selection-Sort
- Insertion-Sort
- Quick-Sort
- Merge-Sort

Die unterschiedlichen Verfahren unterscheiden sich u.a. nach der Effizienz, d.h. nach Bedarf an Speicherplatz und Rechenzeit u. dem Wachstum der Rechenzeit bei steigender Anzahl  $n$  der Eingabeelemente Laufzeitkomplexität)

Beispiel:

Sortialgorithmus A benötigt	$n^2$	Schritte für $n$ Elemente
Sortialgorithmus B benötigt	$n \cdot \log_2(n)$	Schritte für $n$ Elemente

Zur Abschätzung der Aufwandsfunktion  $f(n)$  von Algorithmen wird sehr häufig die **O-Notation** verwendet:

$$f(n) \in O(g(n)) : \Leftrightarrow \exists c, n_0 \forall n \geq n_0 : f(n) \leq c g(n)$$

Das bedeutet, dass ab einem bestimmten  $n$  das  $f(n)$  nicht stärker als ein  $g(n)$  wächst und somit  $g$  als Abschätzung verwendet werden kann.

Daraus ergeben sich zum Beispiel die folgenden Abschätzungsarten:

$O(1)$	konstanter Aufwand	<b>n</b>	<b>log n</b>	<b>n * log(n)</b>	<b>n<sup>2</sup></b>	<b>2<sup>n</sup></b>
$O(\log n)$	logarithmischer Aufwand	2	1	2	4	4
$O(n)$	linearer Aufwand	4	2	8	16	16
$O(n * \log n)$	$n * \log n$ Aufwand	8	3	24	64	256
$O(n^2)$	quadratischer Aufwand	16	4	64	256	65536
$O(n^k)$	polynomialer Aufwand	32	5	160	1024	4,3E+09
$O(2^n)$	exponentieller Aufwand					

Beispiele für die Zusammensetzung von Funktionen und die sich dadurch ergebene Aufwandsabschätzung:

$f(n)$  dominiert  $g(n)$  falls  $f(n)$  bei großen  $n$ -Werten stärker wächst als  $g(n)$ , d.h.

$2^n$  dominiert  $n^k$  für jedes feste  $k$

$n^k$  dominiert  $n^r$ , falls  $k > r$

$n * \log(n)$  dominiert  $n$

$n$  dominiert  $\log_b(n)$  für jede Basis  $b$

Wird  $g(n)$  von  $f(n)$  dominiert, dann gilt

$$O(f(n) + g(n)) = O(f(n))$$

Beispiele:

$$O(n^2 - n) = O(n^2)$$

$$O(5 * n^2 + 88 * n + 1) = O(n^2)$$

$$O(123) = O(1)$$

$$O(n^2 + 3^n) = O(3^n)$$

Wie bestimmt man die Komplexität der Zahl der Schritte, wie schätzt man die Laufzeitkomplexität ab?

zB Maschinenmodell:

- Schritte eines Algorithmus sind zeitkonstante Anweisungen
- Laufzeitunterschiede einzelner Schritte werden vernachlässigt
- Addieren, Runden, Kopieren, bedingte Verzweigung, Aufruf eines Unterprogramms sind jeweils Schritte
- Sortieren ist kein Schritt (sinnvolle Definition von Schritten!)

## Laufzeitkomplexität für einige Sprachkonstrukte:

### elementare Operation

i1=0;	<b>O(1)</b>
-------	-------------

### Sequenz elementarer Operationen

i1=0; i2=0; ... i123=0;	O(1) O(1) ... O(1)	$123 * O(1) = \mathbf{O(1)}$
----------------------------------	-----------------------------	------------------------------

### Schleifen

for (i=0;i<n;i++) a[i]=0;	O(n)	$O(1) * O(n) = \mathbf{O(n)}$
	O(1)	

<pre>for (i=0;i&lt;n;i++) {     a1[i]=0;     ...     a23[i]=0; }</pre>	O (n)	O (n)	O (1) *O (n)= <b>O (n)</b>
	O (1)	23*O (1) = O (1)	
	O (1)		

<pre>for (i=0;i&lt;n;i++)     for (j=0;j&lt;n-1;j++)     {         a1[i][j]=0;         ...         a34[i][j]=0;     }</pre>	O(n)	O(n)*O(n-1) =	O(1)*O(n <sup>2</sup> )= <b>O(n<sup>2</sup>)</b>
	O(n-1)	O(n)*O(n)=O(n <sup>2</sup> )	
	O(1)	34*O(1) = O(1)	
	O(1)		

### bedingte Anweisung

if (x<100) y=x;	O(1)	O(1)	$O(\max\{1, n\}) = \mathbf{O(n)}$
	O(1)		
else for (i=0;i<n;i++) if (a[i]>y) y=a[i];	O(n) O(1) O(1)	$O(n) * O(1) = O(n)$	

Die Laufzeit hängt nicht immer ausschließlich von der Größe des Problems ab, sondern auch von der Beschaffenheit der Eingabemenge.

Daraus ergeben sich

- **beste Laufzeit (best case)**  
beste Laufzeitkomplexität für eine Eingabeinstanz der Größe  $n$
- **schlechteste Laufzeit (worst case)**  
schlechteste Laufzeitkomplexität für eine Eingabeinstanz der Größe  $n$
- **mittlere oder erwartete Laufzeit (average case)**  
gemittelte Laufzeitkomplexität für alle Eingabeinstanzen der Größe  $n$

### Beispiel:

Eingabe:

Feld  $a$  mit  $n$  Elementen mit  $1 \leq a[i] \leq n$

Programmcode:

<pre>if (a[0]==1)     a[0]=1; else     for (i=0; i&lt;n; i++)         a[i]=2;</pre>	$O(1)$	$O(1)$	?
	$O(1)$	$O(1)$	
	$O(n)$ $O(1)$	$O(n)$	

**beste Laufzeit:**  $O(1) + O(1) = O(1)$

**schlechteste Laufzeit:**  $O(1) + O(n) = O(n)$

**mittlere Laufzeit:**

Mittelung der notwendigen Schritte für **alle** Eingabeinstanzen der Größe  $n$

jedes Element  $a[i]$  kann  $n$  Werte annehmen

$n$  Elemente:  $n \cdot n \cdot \dots \cdot n = n^n$  **verschiedene Eingabeinstanzen**

$a[0]=1$  in  $n^{n-1}$  Instanzen

$a[0] \neq 1$  in allen anderen Fällen, also  $n^n - n^{n-1} = n^{n-1} \cdot (n-1)$  Instanzen

Gesamtschritte:

$n^{n-1}$  Instanzen  $\cdot$  1 Schritt +  $n^{n-1} \cdot (n-1)$  Instanzen  $\cdot$   $n$  Schritte

**=  $n^{n-1} + n^n (n-1)$  Schritte**

mittlere Laufzeit: Schritte / Instanzen

$$\frac{n^{n-1} + n^n \cdot (n-1)}{n^n} = \frac{1}{n} + n - 1 = O(n)$$

## 14.2 Suchalgorithmen

### Beispiele für Suchalgorithmen:

- **lineare/sequentielle Suche**
- **binäre Suche**
- weitere Suchverfahren auf sortierten Feldern
  - Fibonacci-Suche
  - Sprung-Suche
  - Exponentielle Suche

### Aufgabenstellung:

In einem Telefonbuch sollen Einträge gesucht werden. Die Datensätze bestehen aus

Nachname

Vorname

Adresse und

Telefonnummer,

Sie sind geordnet nach Nachname, bei Gleichheit nach Vorname.

Nach- und Vorname sind Schlüssel.

**Aufgabe 1:** Wenn man den Namen kennt und die Telefonnummer wissen möchte, findet man den Eintrag relativ rasch:

- Aufschlagen des Telefonbuchs in der Mitte
- Dann wird entschieden, ob man in der linken oder in der rechten Hälfte weitersucht  
→ **binäre Suche**

**Aufgabe 2:** Das Telefonbuch von Wien ist 100-mal so dick, wie das vorhin durchsuchte. Dauert das Suchen auch 100-mal so lange?

- Nein- für die Suche werden nur 6-7 weitere Namen gelesen/verglichen werden.  
→ **binäre Suche ist effizient**
- Der **Aufwand der binären Suche** ist proportional  **$\log_2(n)$** , dem binären Logarithmus von  $n$

**Aufgabe 3:** Wer hat die Telefonnummer 52575 ?

- Die Information ist auch im Telefonbuch, aber nur schwer zu finden
- jeder Datensatz muss durchsucht werden.  
→ **lineare/sequentielle Suche**

**Aufgabe 4:** Wie lange dauert die Suche in einem 100-mal dickeren Telefonbuch?

- Die Suche dauert 100-mal so lange.
- Der **Aufwand der linearen Suche** ist proportional der Anzahl  $n$  der Datensätze



## 14.2.1 sequentielle Suche

**Prinzip:** überprüfe sequentiell alle Elemente des Feldes (brute force search)

```
int seqSearch(int k, int a[], int n)
{
    int i=0;
    while (i<n && a[i]!=k)           //Zwei Abfragen pro Durchlauf
    {
        i++;
    }
    if (i<n)
        return i;                   //Element gefunden
    else
        return -1;                  //Element nicht gefunden
}
```

optimierte Variante:

Man fügt das gesuchte Element an die erste (oder letzte) Stelle des Feldes ein. Das spart eine Abfrage pro Durchlauf:

```
int seqSearch(int k, int a[], int n)
{
    int i = n;                       //durchsucht a[1] .. a[n] nach k
    a[0] = k;                       //a[0] gehört nicht zur durchsuchenden Menge
                                    //(= Stopp-Element)

    do {
        i--;
    } while (a[i]!=k);               //Eine Abfrage pro Durchlauf
    if (i!=0)
        return i;
    else
        return -1;
}
```

**Laufzeit:**

- bester Fall:  $O(1)$
- schlechter Fall:  $O(n)$
- mittlerer Fall:  $\frac{1}{n} \sum_{i=1}^n i = \frac{1}{n} \frac{n \cdot (n+1)}{2} = \frac{n+1}{2} = O(n)$

## 14.2.2 binäre Suche

Ist die zu durchsuchende Menge von Datensätzen sortiert, kann die Suche beschleunigt werden, indem das gesuchte Element mit dem mittleren Element des Feldes verglichen und bei Ungleichheit nur jeweils die linke oder die rechte Hälfte des Feldes rekursiv weiter betrachtet wird.

### Prinzip:

Suche von  $k$  auf einem sortierten Feld  $a$



Vergleiche  $k$  mit  $a[m]$  wobei  $m = (0+n)/2$

Fall 1:  $k == a[m]$  → fertig

Fall 2:  $k < a[m]$  → rekursive Suche von  $k$  in  $a[0] \dots a[m-1]$

Fall 3:  $k > a[m]$  → rekursive Suche von  $k$  in  $a[m+1] \dots a[n]$

### iterative Variante:

```
int binSearch1(int k, int a[], int l, int r)
{
    int m;

    while (l <= r)
    {
        m = (l+r) / 2;
        if (k == a[m]) return m;
        if (k < a[m]) r = m-1;
        if (k > a[m]) l = m+1;
    }
    return -1;
}
```

### rekursive Variante:

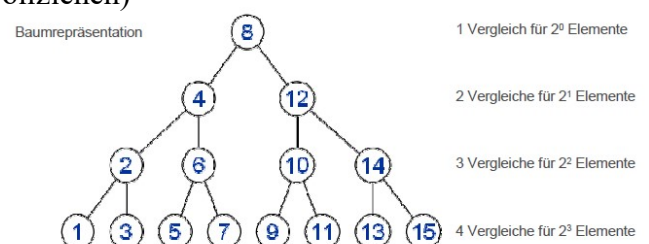
```
int binSearch2(int k, int a[], int l, int r)
{
    int m;

    if (l > r) return -1;

    m = (l+r) / 2;
    if (k == a[m]) return m;
    if (k < a[m]) return binSearch2(k, a, l, m-1);
    if (k > a[m]) return binSearch2(k, a, m+1, r);
}
```

### Laufzeit:

- jeder Vergleich reduziert die zu durchsuchende Menge um den Faktor 2
- schlechtester Fall:  $O(\log n)$
- bester Fall:  $O(1)$
- mittlerer Fall:  $\frac{\log(n+1) \cdot (n+1) - (n+1) + 1}{n} \approx \log(n+1) - 1$  ergibt  $O(\log n)$   
Im Durchschnitt 1 Vergleich weniger als die Maximalzahl möglicher Vergleiche  
(intuitiv durch Baumrepräsentation nachzuvollziehen)



## 14.3 Sortialgorithmen

### Beispiele für Sortiervverfahren

- Bubble-Sort
- Selection-Sort
- Insertion-Sort
- Quick-Sort
- Merge-Sort
- Heap-Sort
- Shell-Sort

### Sortialgorithmen unterscheiden sich u.a. hinsichtlich folgender Eigenschaften

- Effizienz (best, average, worst case)
- Speicherbedarf
  - in-place (zusätzlicher Speicher von der Eingabegröße unabhängig)
  - out-of-place (zus. Speicherbedarf von der Eingabegröße abhängig)
- rekursiv oder iterativ
- Stabilität: stabile Verfahren verändern die Reihenfolge von gleichen Elementen nicht

#### 14.3.1 Bubble-Sort

##### Prinzip

- durchlaufe die Menge und vertausche zwei aufeinanderfolgende Elemente, wenn ihre Reihenfolge nicht stimmt.
- durchlaufe die Menge gegebenenfalls mehrmals, bis bei einem Durchlauf keine Vertauschungen mehr durchgeführt werden mussten.

```
void bubblesort(int a[], int gr)
{
    int sortiert, i, tmp;
    do {
        sortiert = 1;
        for (i=1; i<gr; i++)
        {
            if (a[i-1]>a[i])
            {
                tmp=a[i-1];
                a[i-1]=a[i];
                a[i]=tmp;
                sortiert=0;
            }
        }
    } while (!sortiert);
}
```

##### Eigenschaften

- iterativ, nicht rekursiv
- stabil (gleiche benachbarte Schlüssel werden nicht getauscht)
- in-place (konstanter zusätzlicher Speicheraufwand)
- effizient für vorsortierte Mengen

##### Laufzeit

- bester Fall: Eingabe sortiert, ein Durchlauf:  $O(n)$
- schlechtester Fall: Eingabe ist umgekehrt sortiert:  $O(n^2)$
- durchschnittlicher Fall:  $O(n^2)$

### 14.3.2 Selection-Sort

#### Prinzip

- durchlaufe die Menge und finde das kleinste Element
- vertausche das kleinste Element mit dem ersten Element
- vorderer Teil ist sortiert (k Elemente nach Durchlauf k), hinterer Teil ist unsortiert (n-k) Elemente
- durchlaufe die hintere, nicht sortierte Teilmenge und finde das kleinste Element
- vertausche das n-kleinste Element mit dem n-ten Element

Bsp:

55	7	78	12	42
7	55	78	12	42
7	12	78	55	42
7	12	42	55	78

```
int minimum (int a[],int anfang,int ende)
{
    int index;
    int minIdx = anfang;
    for (index=anfang+1; index<=ende; index++)
    {
        if (a[index] < a[minIdx])
            minIdx = index;
    }
    return minIdx;
}
```

```
void selectionSort (int a[], int gr) {
    int index, minIdx;
    int tmp;
    for (index=0; index<gr-1; index++) {
        minIdx = minimum(a,index,gr-1);

        //vertausche a[index] mit a[minIdx]
        tmp=a[index];
        a[index]=a[minIdx];
        a[minIdx]=tmp;
    }
}
```

#### Eigenschaften

- iterativ, nicht rekursiv
- stabil oder instabil (je nach Implementierung)
- in-place (konstanter zusätzlicher Speicheraufwand)

#### Laufzeit

- bester Fall, mittlerer und schlechtester Fall:  $O(n^2)$

### 14.3.3 Insertion-Sort

#### Prinzip

- erstes Element ist sortiert, hinterer Teil mit  $n-1$  Elementen ist unsortiert
- entnehme der hinteren, unsortierten Menge ein Element und füge es an die richtige Position der vorderen, sortierten Menge ein ( $n-1$  mal)
- Einfügen in die vordere, sortierte Menge erfordert das Verschieben von Elementen

Bsp:

5	2	4	6	1	3	Elemente 1..1 sind sortiert, 2..n unsortiert
5	2	4	6	1	3	Vergleiche 2 mit allen Elementen der sortierten Menge beginnend mit dem größten. Wenn ein Element größer als 2 ist, schiebe es eins nach rechts, sonst füge 2 ein.
2	5	4	6	1	3	Elemente 1..2 sind sortiert, 3..n unsortiert
2	5	4	6	1	3	Vergleiche 4 mit allen Elementen der sortierten Menge. Wenn ein Element größer als 4 ist, verschiebe es.
2	4	5	6	1	3	Elemente 1..3 sind sortiert. Einfügen von 6.
2	4	5	6	1	3	Elemente 1..4 sind sortiert. Einfügen von 1 (Dazu werden Elemente 6,5,4 und 2 jeweils um eins nach rechts verschoben. Danach wird 1 an Pos. eins eingefügt.
1	2	4	5	6	3	....
1	2	3	4	5	6	

```
int insertionSort (int a[], int gr)
{
    int j,i;
    int value;
    for (j=1;j<gr;j++)
    {
        value = a[j];
        i = j-1;
        for (i=j-1; a[i]>value && i>=0; i--)
        {
            a[i+1] = a[i];
        }
        a[i+1] = value;
    }
}
```

#### Eigenschaften

- iterativ, nicht rekursiv
- stabil (gleiche benachbarte Schlüssel werden nicht getauscht)
- in-place (konstanter zusätzlicher Speicheraufwand)
- effizient für vorsortierte Mengen

#### Laufzeit

- bester Fall: Menge ist sortiert  $O(n)$
- schlechter Fall: Menge ist umgekehrt sortiert:  $O(n^2)$
- mittlerer Fall:  $O(n^2)$

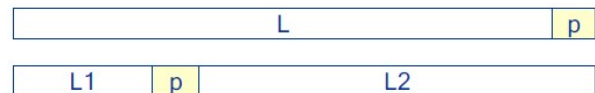
## 14.3.4 Quick-Sort

### Prinzip

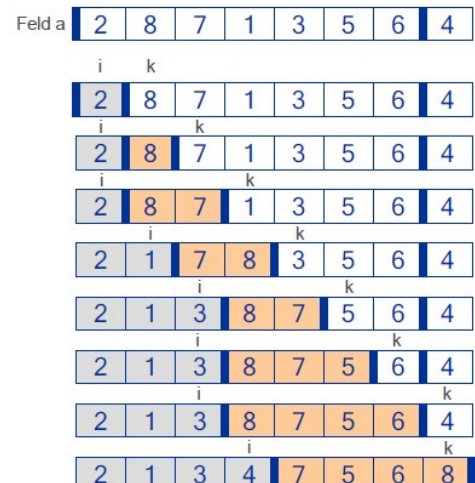
- Teile-und-Herrsche-Ansatz
- Mengen mit einem oder keinem Element sind sortiert
- ansonsten:
  - Aufteilung in Teilmengen:
    - Wahl eines Schlüssels/Pivotelements
    - Elemente, die kleiner als das Pivotelement sind, werden der linken Teilmenge zugeordnet
    - Elemente, die größer als das Pivotelement sind, werden der rechten Teilmenge zugeordnet
  - rekursiver Aufruf des Algorithmus für beide Teilmengen
- Verbindung der Teilergebnisse (automatisch, da Sortierung der Teilmengen in-place)

### Pseudocode QuickSort (L):

```
wenn L nur 1 Element besitzt  
dann { return; }  
sonst { wähle Pivotelement p aus L;  
        L1 = { a in L | a < p };  
        L2 = { a in L | a > p };  
        QuickSort (L1);  
        QuickSort (L2); }
```



```
int aufteilung (int a[], int l, int r)  
{  
    int tmp, k;  
    int pivot = a[r]; //gewähltes Pivotelements  
    int i = l - 1;  
  
    for (k = l; k <= r-1; k++)  
    {  
        if (a[k] <= pivot)  
        {  
            i++;  
            //tausche a[i] mit a[k]  
            tmp = a[i];  
            a[i] = a[k];  
            a[k] = tmp;  
        }  
    }  
    i++;  
    //tausche a[i] mit a[k]  
    tmp = a[i];  
    a[i] = a[k];  
    a[k] = tmp;  
  
    return i;  
}
```



```
void quickSort (int a[], int l, int r)  
{  
    int i;  
    if (l < r) {  
        i = aufteilung (a, l, r);  
        quickSort (a, l, i-1);  
        quickSort (a, i+1, r);  
    }  
}
```

## Eigenschaften

- rekursiv
- in-place (konstanter zusätzlicher Speicheraufwand)
- nicht stabil

## Laufzeit

- bester Fall:  
Aufteilung von n-Elementen in 2 gleich große Teilfolgen  $(n/2) + (n/2)$   
Rekursionstiefe von  $\log n \rightarrow O(n * \log n)$
- schlechtester Fall:  
Aufteilung von n Elementen in 2 ungleiche Teilfolgen  $(n-1) + (1)$   
Rekursionstiefe von  $n \rightarrow O(n^2)$
- mittlerer Fall:  $O(n * \log n)$

## 14.3.5 Merge-Sort

### Prinzip

- Teile-und-Herrsche-Ansatz
- Mengen mit einem oder keinem Element sind sortiert
- ansonsten:
  - Aufteilung des Problems in zwei gleich große Teilmengen
  - rekursiver Aufruf des Algorithmus für beide Teilmengen
  - Verbindung der Teilmengen
- im Vergleich zu Quick-Sort:
  - simple Unterteilung im Vergleich zu Quick-Sort
  - optimale Rekursionstiefe von  $\log n$
  - aufwendiger Merge-Schritt

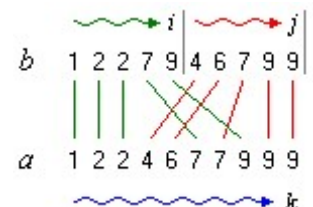
```
// einfache Variante
void merge(int a[], int l, int m, int r)
{
    int i, j, k;
    int gr=r+1;
    int *b; //Hilfsarray b
    if ((b=(int *)malloc(sizeof(int)*gr))==NULL) return;

    // beide Hälften von a in Hilfsarray b kopieren
    for (i=l; i<=r; i++)
        b[i]=a[i];

    i=l; j=m+1; k=l;
    // jeweils das nächstgrößte Element zurückkopieren
    while (i<=m && j<=r)
        if (b[i]<=b[j])
            a[k++]=b[i++];
        else
            a[k++]=b[j++];

    // Rest der vorderen Hälfte falls vorhanden zurückkopieren
    while (i<=m)
        a[k++]=b[i++];

    free(b);
}
```



```

void mergeSort ( int a[], int l, int r )
{
    int m;

    if (l < r)
    {
        m = (l+r) / 2;
        mergeSort(a,l,m);
        mergeSort(a,m+1,r);
        merge(a,l,m,r);
    }
}

```

### Eigenschaften

- rekursiv
- je nach Implementierung in-place oder zusätzlicher Speicheraufwand
- stabil
- im Gegensatz zu Quick-Sort immer garantierte Aufteilung in 2 gleichgroße Teilfolgen

### Laufzeit

- bester, mittlerer, schlechterer Fall:  $O(n * \log n)$

## 14.3.6 Zusammenfassung

Verfahren	bester Fall $O(x)$	mittlerer Fall $O(x)$	schlechtester Fall $O(x)$	stabil	rekursiv	Speicher $O(x)$
Bubble	n	$n^2$	$n^2$	ja	nein	1
Selection	$n^2$	$n^2$	$n^2$	ja	nein	1
Insertion	n	$n^2$	$n^2$	ja	nein	1
Quick	$n \log n$	$n \log n$	$n^2$	nein	ja	1
Merge	$n \log n$	$n \log n$	$n \log n$	ja	ja	n



## 15 Dynamische Datenstrukturen

Bei dynamischen Datenstrukturen handelt es sich um Speichermechanismen, die sich flexibel dem momentanen Speicherplatzbedarf eines Programms anpassen. Bei einem Array muss die Größe zu Beginn definiert werden und kann nicht mehr verändert werden. Das verschwendet entweder Speicherplatz oder erzeugt einen Fehler, wenn das Array überläuft, weil es voll ist!

### 15.1 Einfach verkettete Listen

#### 15.1.1 Anwendungsgebiete

Wofür kann diese Datenstruktur eingesetzt werden? Prädestiniert ist sie als Stack und (in nachfolgender Implementierung mit der Referenz listEnd) als Queue.

Ein Stack ist eine FILO (First In, Last Out) Datenstruktur. Das bedeutet, dass das erste Datenelement, das gespeichert wird, das letzte ist, das wieder aus der Datenstruktur herausgenommen werden kann. Man kann sich das, wie der Name schon andeutet, als einen Stapel z.B. Bücher auf einem Tisch vorstellen. Das Buch, das man als erstes auf die Tischplatte gelegt hat, bekommt man erst (ohne Probleme) wieder, wenn all die Bücher, die man auf dieses gestapelt hatte, wieder entfernt wurden. Der Stack ist in der Informatik eine sehr wichtige Datenstruktur.

Er wird z.B. im Bereich des Compilerbaus und, während der Ausführung eines Programms, bei jeder Parameterübergabe an eine Funktion / Prozedur / Methode eingesetzt. Die Implementierung mit Hilfe einer linearen Liste sieht so aus: Neue Elemente werden vorne in die Liste eingefügt und auch von dort wieder weggenommen (lesen und löschen des ersten Knotens in der Liste).

Die Queue ist das Gegenstück zum Stack, eine FIFO (First In, First Out) Datenstruktur. Sie wird z.B. als Puffer für eintreffende Nachrichten verwendet: die erste Nachricht, die in den Puffer geschrieben wird, ist auch die erste, die aus diesem gelesen wird. Bei der Implementierung würde am nächsten liegen, neue Datenelemente vorne in die Liste einzufügen und am Ende zu entnehmen.

Ein weiteres Einsatzgebiet besitzt die lineare Liste überall dort, wo Daten zunächst einmal gesammelt und dann immer wieder komplett durchlaufen werden müssen.

#### 15.1.2 Aufbau

Einfach verkettete Listen erzeugt man, indem man eine Struktur für ein einzelnes Listenelement erschafft und diese dann je nach Bedarf aneinander hängt.

Zuerst benötigt man also eine Struktur in C für ein solches Listenelement:

```
typedef struct LISTELEMENT{
    int data;
    struct LISTELEMENT *next;
}ListElement;
```

Die Struktur trägt den Namen LISTELEMENT und durch den Befehl typedef erzeugen wir aus dieser Struktur den Datentyp ListElement. Dieser enthält nur zwei Felder, nämlich eines, das die Daten trägt, die gespeichert werden sollen und ein weiteres Feld, das ein Pointer auf die Struktur LISTELEMENT ist.

Grafisch sieht eine solche Liste dann wie folgt aus:

### Lineare Liste

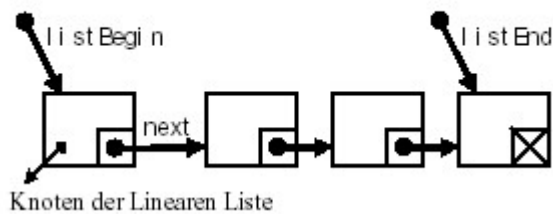


Abbildung: Einfach-verkettete Liste

Das erste Element enthält Daten und einen Zeiger auf das folgende Element. So geht das immer weiter, bis schließlich das Ende der Liste erreicht wird. Hier befindet sich ein Element vom Typ ListElement, bei dem next auf NULL initialisiert wurde, um das Ende der Liste zu markieren.

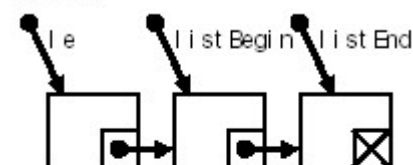
### 15.1.3 insertElement()

Diese Methode gehört wohl zu den Grundfunktionen einer Datenstruktur. Für die lineare Liste bedeutet diese Operation, dass ein neuer Knoten am Beginn der Liste eingefügt wird. Der Aufbau von insertElement() ist relativ einfach. Das in dem Knoten zu speichernde Datenelement wird der Methode als Parameter übergeben (int data). Jetzt wird ein neues ListElement benötigt, da ja ein neuer Knoten in die Liste eingehängt werden soll (malloc()). Danach folgen die Operationen zum Einhängen des neuen Knotens in die Liste. Dessen Referenz next muss auf den bisherigen Listenbeginn verweisen. Mit der Aktualisierung von listBegin ist das neue Listenelement korrekt eingefügt.

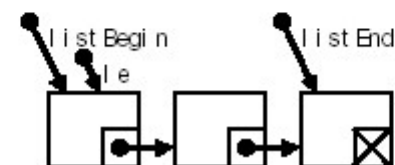
#### Normalfall: Schritt 1



#### Schritt 2



#### Schritt 3



```
void insertElement(int data, ListElement **listB, ListElement **listE)
{
    ListElement *le=NULL;

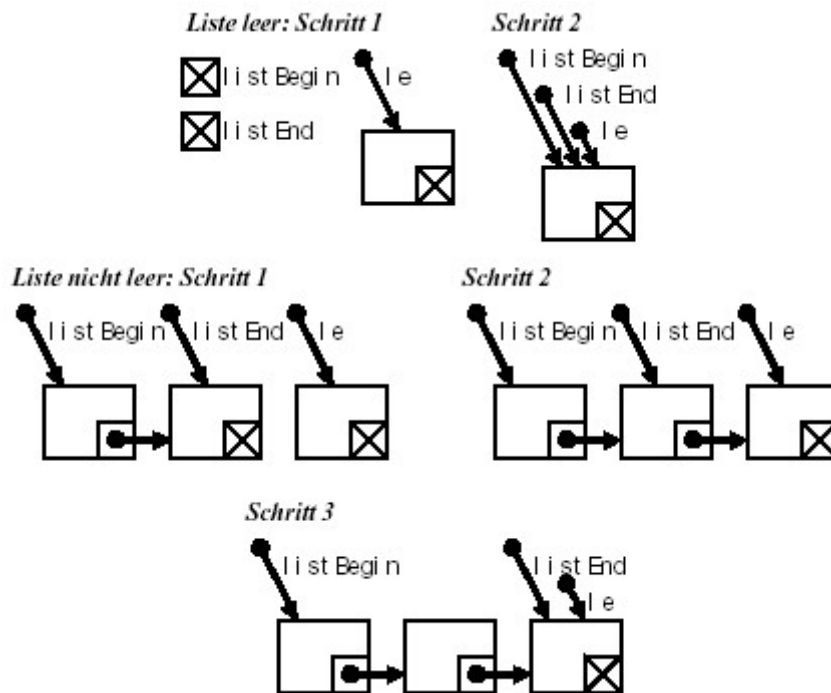
    if ((le=(ListElement *)malloc(sizeof(ListElement)))==NULL) {
        fprintf(stderr,"Nicht genugend Speicher verfuegbar!"); return ;
    }

    le->data=data;
    le->next=*listB;
    *listB=le;

    if (*listE==NULL){
        *listE=le;
    }
}
```

### 15.1.4 appendElement()

Mit dieser Funktion wird ein neuer Knoten am Ende der linearen Liste angehängt. Diese Operation wird z.B. bei der Implementierung einer Queue benötigt.



Quellcode:

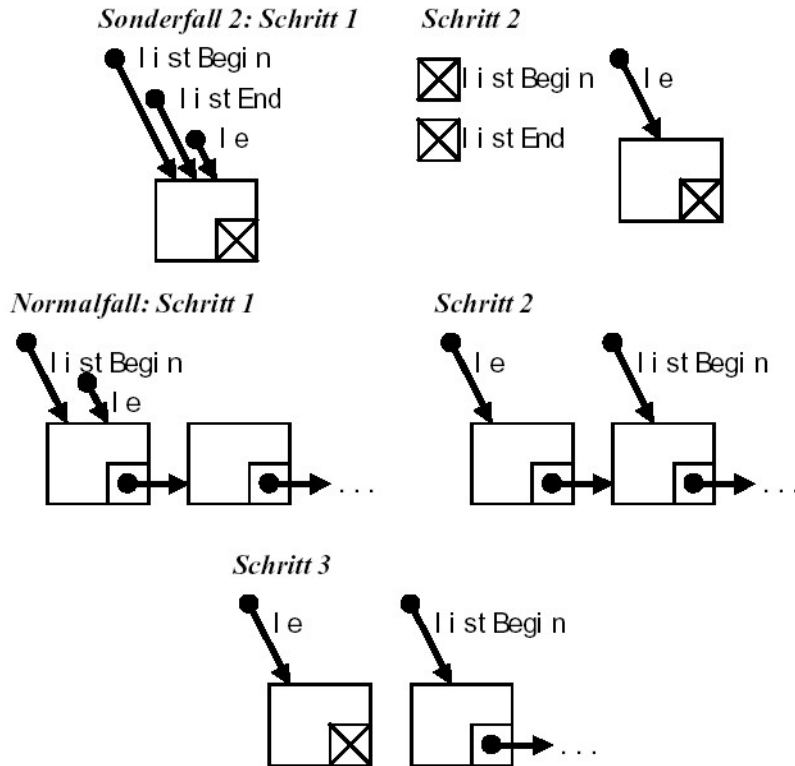
```
void appendElement(int data, ListElement **listB, ListElement **listE){  
  
    ListElement *le=NULL;  
  
    if ((le=(ListElement *)malloc(sizeof(ListElement)))==NULL){  
        fprintf(stderr,"Nicht genug Speicher verfuegbar!"); return;  
    }  
    le->data=data;  
    le->next=NULL;  
  
    if (*listB==NULL){  
        *listB=le;  
        *listE=le;  
    }  
    else{  
        (*listE)->next=le;  
        *listE=le;  
    }  
}
```

### 15.1.5 deleteFirstElement()

Grundsätzlich benötigt man, falls man einen Stack oder eine Queue implementieren möchte, nur die Entfernung des ersten Listenelementes zu programmieren.

Sonderfälle:

1. die Liste ist leer
2. es befindet sich nur ein Element in der Liste
3. Normalfall

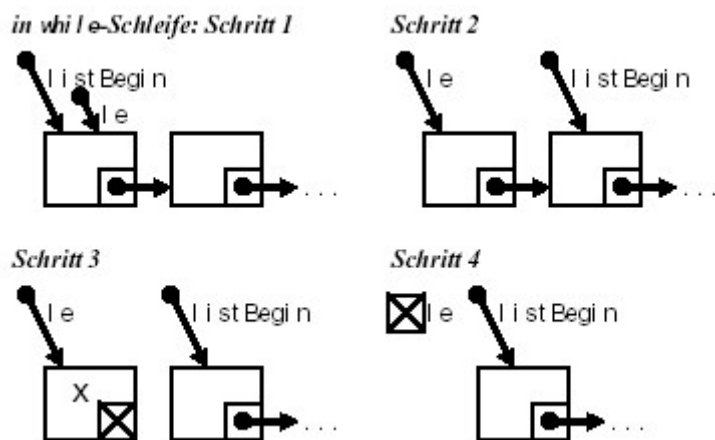


```
int deleteFirstElement(int *data, ListElement **listB, ListElement
**listE){
    ListElement *temp;

    // Elementarprüfung: Ist die Liste leer?
    if (*listB==NULL){
        fprintf(stderr,"%s","Fehler: Liste ist noch leer\n");
        *listE=NULL;
        *listB=NULL;
        return EXIT_FAILURE;
    }
    //ist nur ein Element in der Liste?
    if ((*listB)->next==NULL){
        *data=(*listB)->data;
        free(*listB);
        *listE=NULL;
        *listB=NULL;
    }
    else{ // 1.Element übergeben, Speicher freigeben
        *data=(*listB)->data;
        temp=(*listB)->next;
        free(*listB);
        *listB=temp;
    }
    return EXIT_SUCCESS;
}
```

### 15.1.6 ClearAllElements()

Diese Funktion dient zum vollständigen Entfernen der Liste aus dem Speicher.



```
void ClearAllElements(ListElement **listBegin, ListElement **listEnd){  
    ListElement *le;  
  
    while(*listBegin!=NULL){  
        le=(*listBegin)->next;  
        free(*listBegin);  
        *listBegin=le;  
    }  
    *listEnd=NULL;  
}
```

### 15.1.7 deleteElement()

In dem nachfolgenden Programmfragment wird allerdings auch die beliebige Entfernung eines innerhalb der Liste vorkommenden Elementes gezeigt.

Wenn man ein Element innerhalb der Liste löschen möchte, so muss man einfach dafür sorgen, dass dieses Element in der Liste übersprungen wird.

Man setzt also den Zeiger des Vorgängerelementes auf das Nachfolgeelement. Das zu löschende Element besitzt zwar noch einen Zeiger auf seinen Nachfolger, das spielt aber keine Rolle, da auf das Element selbst kein Zeiger mehr existiert, es also nie erreicht werden kann.

Allerdings gilt es einige Sonderfälle zu beachten:

1. die Liste ist leer
2. es befindet sich nur ein Element in der Liste
3. soll das erste Element gelöscht werden
4. soll das letzte Element gelöscht werden
5. Normalfall

```

int deleteElement(int *data, int index, ListElement **listB, ListElement
**listE){
    ListElement *temp, *prev;

    // Elementarprüfung: Ist die Liste leer?
    if (*listB==NULL){

        *listE=NULL;
        /*listB=NULL;
        return EXIT_SUCCESS;

    }
    //ist nur ein Element in der Liste?
    if ((*listB)->next==NULL){
        *data=(*listB)->data;
        free(*listB);
        *listE=NULL;
        *listB=NULL;
    }
    else{ // Elementarprüfung: 1.Element löschen?
        if (index==0){
            // 1.Element übergehen, Speicher freigeben
            *data=(*listB)->data;
            temp=(*listB)->next;
            free(*listB);
            *listB=temp;
        }
        else{
            temp=*listB;

            // Liste durchlaufen bis Index oder Ende
            while ((temp->next != NULL)&&(index != 0)){
                // Vorgängerelement merken und weitergehen
                prev = temp;
                temp = temp->next;
                index--;
            }
            // Index nicht gefunden -> Liste zu kurz?
            if (index != 0){
                fprintf(stderr,"%s","Fehler: Liste zu kurz\n");
                return EXIT_FAILURE;
            }
            // Element übergehen, Speicher freigeben
            *data=temp->data;
            prev->next = temp->next;
            //Letztes Element gelöscht
            if (temp->next==NULL){
                *listE=prev;
            }
            free(temp);
        }
    }

    return EXIT_SUCCESS;
}

```

### 15.1.8 dataOutput()

Das folgende Programmfragment soll der Vollständigkeit dienen und zeigt die einfache Ausgabe der gesamten Liste.

```
void dataOutput(ListElement *listB, ListElement *listE){  
  
    ListElement *le=NULL;  
  
    le=listB;  
    while (le!=NULL){  
        printf("%d \n", le->data);  
        le=le->next;  
    }  
}
```

### 15.1.9 Beispiel

Folgender Ausdruck soll ausgewertet werden:

$5 * (((9 + 8) * (4 * 6)) + 7)$

```
push(5);  
push(9);  
push(8);  
push(pop+pop);  
push(4);  
push(6);  
push(pop*pop);  
push(pop*pop);  
push(7);  
push(pop+pop);  
push(pop*pop);
```

//ausgabe;

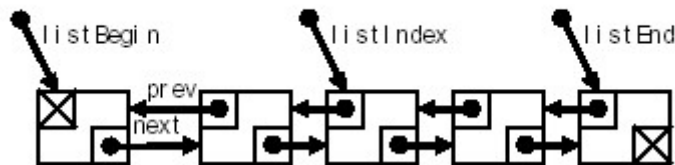
entspricht dem Ausdruck:  $598 + 46 * 7 + *$

## 15.2 Doppelt verkettete Liste

### 15.2.1 Aufbau

Der Aufbau der doppelt verketteten linearen Liste ähnelt sehr dem der einfach verketteten. Bei jedem Listenknoten kommt lediglich eine neue Referenz (prev ) hinzu.

Doppelt verkettete Lineare Liste



Damit ist es nun möglich, sich nicht nur vorwärts, sondern auch rückwärts durch die Liste zu bewegen.

Anwendung: Benötigt man einen Datenspeicher, in dem man beliebig blättern kann, so ist die doppelt verkettete lineare Liste eine gute Wahl. Weiters bietet sie Vorteile, wenn man eine sortierte Liste erstellen wird und dabei bereits teilweise sortierte Daten einfügen möchte.

### 15.2.2 insertElementBefore()

Diese Funktion dient zum Einfügen eines Listenelements **vor** einem Knoten, der als Parameter der Funktion übergeben wird. Dabei sind folgende Anwendungsfälle zu beachten:

1. Liste ist leer
2. Einfügen eines Elements am Beginn der Liste
3. Normalfall

```
int insertElementBefore(int data, ListElement *index, ListElement **listB,
ListElement **listE){
    ListElement *le=NULL;
    if ((le=(ListElement *)malloc(sizeof(ListElement)))==NULL){
        fprintf(stderr,"Nicht genug Speicher verfuegbar!");
        return EXIT_FAILURE;
    }
    le->data=data;

    /* Spezialfall 1: Liste ist leer */
    if ((*listB==NULL) && (index==NULL)){

        *listE=le;
        *listB=le;
        le->next=NULL;
        le->prev=NULL;
        return EXIT_SUCCESS;
    }

    /*Spezialfall 2: Einfügen am Beginn der Liste*/
    if (*listB==index){
        (*listB)->prev=le;
        le->next=(*listB);
        *listB=le;
        le->prev=NULL;
        return EXIT_SUCCESS;
    }
}
```



```

    /* Normalfall */
    le->prev=index->prev;
    index->prev=le;
    le->next=index;
    (le->prev)->next=le;
    return EXIT_SUCCESS;
}

```

### 15.2.3 insertElementBegin()

Diese Implementierung wird zum Einfügen eines neuen Listenelementes am Beginn der DvL verwendet. Da die vorige Funktion diesen Teil eigentlich schon abgedeckt hat, kann sie bequem für diese Funktion wieder verwendet werden.

```

int insertElementBegin(int data, ListElement **listB, ListElement **listE){

    if (insertElementBefore(data, *listB, listB, listE))
        return EXIT_FAILURE;
    return EXIT_SUCCESS;

}

```

### 15.2.4 insertElementAfter()

Der Aufbau dieser Funktion, die ein neues Listenelement hinter einem gewünschten Knoten einfügt, ist dual zur vorigen Funktion und bedarf keiner weiteren Erläuterung.

```

int insertElementAfter(int data, ListElement *index, ListElement **listB,
ListElement **listE){

    ListElement *le=NULL;

    if ((le=(ListElement *)malloc(sizeof(ListElement)))==NULL){
        fprintf(stderr,"Nicht genugend Speicher verfuegbar!");
        return EXIT_FAILURE;
    }

    le->data=data;
    /* Spezialfall 1: Liste ist leer */
    if ((*listB==NULL) && (index==NULL)){

        *listE=le;
        *listB=le;
        le->next=NULL;
        le->prev=NULL;
        return EXIT_SUCCESS;

    }
    /*Spezialfall 2: Einfügen am Ende der Liste*/

    if (*listE==index){
        (*listE)->next=le;
        le->prev=*listE;
        *listE=le;
        le->next=NULL;
        return EXIT_SUCCESS;
    }
}

```

```

/* Normalfall */
le->prev=index;
le->next=index->next;
index->next=le;
(le->next)->prev=le;
return EXIT_SUCCESS;
}

```

### 15.2.5 appendElement()

Wie unschwer zu erraten ist, dient diese Funktion zum Anhängen eines neuen Listenelementes.

```

int appendElement(int data, ListElement **listB, ListElement **listE){

    return (insertElementAfter(data,*listE,listB,listE));

}

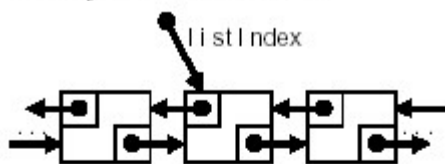
```

### 15.2.6 deleteElement()

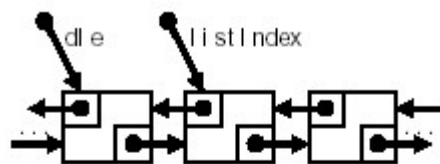
Bei der komplexesten Funktion der DvL gibt es eine Unmenge von Sonderfällen zu beachten:

1. ist die Liste noch leer
2. ist nur ein Element in der Liste?
3. Elementarprüfung: 1.Element löschen?
4. Elementarprüfung: Letztes Element löschen?
5. Normalfall

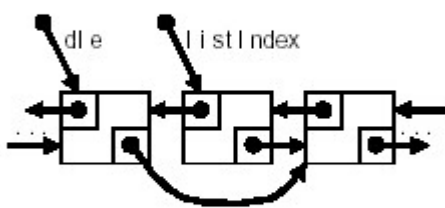
*Normalfall (FORWARD): Schritt 1*



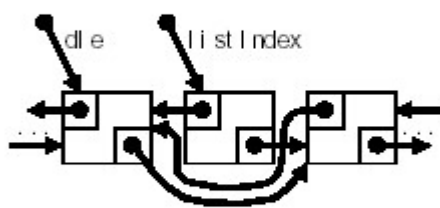
*Schritt 2*



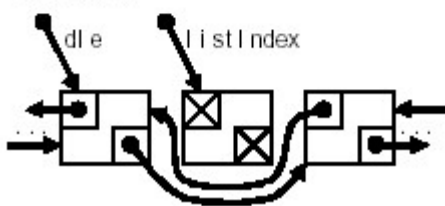
*Schritt 3*



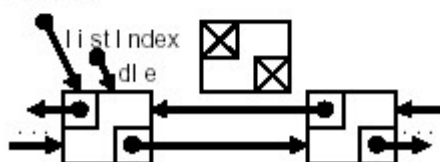
*Schritt 4*



*Schritt 5 & 6*



*Schritt 7*



```

int deleteElement(int *data, ListElement **index, ListElement **listB,
ListElement **listE){

    // Elementarprüfung: Ist die Liste leer?
    if (*listB==NULL){

        *listE=NULL;
        *index=NULL;
        /*listB=NULL;
        return EXIT_FAILURE;

    }
    //ist nur ein Element in der Liste?
    if ((*listB)->next==NULL) && (*listB==*index)){
        *data=(*listB)->data;
        free(*listB);
        *listE=NULL;
        *listB=NULL;
        *index=NULL;
        return EXIT_SUCCESS;
    }
    // Elementarprüfung: 1.Element löschen?
    if (*listB==*index){
        // 1.Element übergehen, Speicher freigeben
        *data=(*listB)->data;
        *listB=(*listB)->next;
        (*listB)->prev=NULL;
        free(*index);
        *index=NULL;
        return EXIT_SUCCESS;
    }
    // Elementarprüfung: Letztes Element löschen?
    if (*listE==*index){
        *data=(*listE)->data;
        *listE=(*listE)->prev;
        (*listE)->next=NULL;
        free(*index);
        *index=NULL;
        return EXIT_SUCCESS;
    }

    ((*index)->prev)->next=(*index)->next;
    ((*index)->next)->prev=(*index)->prev;
    free(*index);
    *index=NULL;
    return EXIT_SUCCESS;
}

```

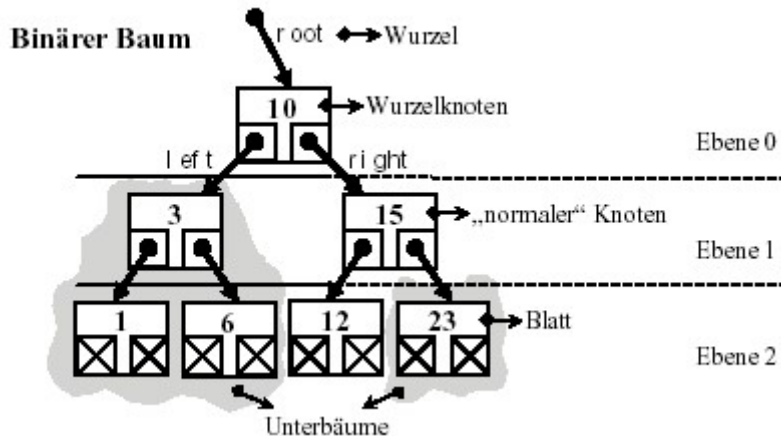
### 15.2.7 dataoutput() und clearAllElements()

siehe einfach verkettete Liste

## 15.3 Der binäre Baum

### 15.3.1 Begriffsbestimmungen

Die linearen Listen sind einfache und relativ vielseitige Datenstrukturen. Doch bei einer gar nicht unwichtigen Funktion versagen sie kläglich: **beim Suchen** eines gespeicherten Datenelements. Das nun ist die Stärke des binären Baums.



Die Grafik zeigt den Aufbau eines binären Baumes und bereits einige damit verbundene, wichtige Begriffe.

Zunächst zur Struktur des Baumes. Wie auch bei den linearen Listen benötigt man eine Referenz, von der aus auf die Datenstruktur zugegriffen werden kann. Bei den Listen war dies `listBegin` (und zusätzlich noch `listEnd`), bei Bäumen (ganz allgemein) wird diese Referenz meist **root**, die **Wurzel**, genannt. Ist der Baum nicht leer, so verweist `root` auf den sogenannten Wurzelknoten.

Jeder Knoten besitzt Daten (`data`). Zusätzlich besitzt der binäre Baum zwei weitere Referenzen (daher der Name: binär/zwei), hier **left** und **right** genannt, die ihrerseits wieder auf weitere Knoten des Baumes verweisen können.

Vielleicht hat man schon erkannt, dass eine Sortierung der Daten durchgeführt wird (erst diese macht überhaupt eine effiziente Suche möglich). Ausgehend vom Wurzelknoten, dessen Datenelement in der Grafik die Größe 10 besitzt, liegen links davon alle Elemente, die kleiner sind, und rechts alle diejenigen, die größer sind. Da drängt sich gleich eine Frage auf: Was passiert mit einem Datenelement, das die gleiche Größe besitzt? Hier ist es Geschmacksache, wohin dieses gesteckt wird. Bei unserer Implementierung wird es nach rechts geschoben.

Im Zusammenhang mit Bäumen als Datenstruktur gibt es noch einige weitere Begriffe, die man kennen sollte.

Wohl einer der wichtigsten ist der des **Unterbaums** (in der Grafik sind einige Beispiele grau unterlegt). Betrachtet man die Referenz `left` des Wurzelknotens, so könnte man diese auch als `root` eines kleineren binären Baumes sehen, der dann den Wurzelknoten mit Größe 3 besitzt. Die Bedingungen für einen Baum sind erfüllt: alle kleineren Datenelemente liegen links, die größeren rechts. Auch der einzelne Knoten mit dem Datenelement der Größe 23 ist natürlich ein Unterbaum (des Baumes mit Wurzelement 15, der wiederum Unterbaum des Gesamtbaumes ist), einer, der eben nur ein Element enthält.

So wie den Begriff **Wurzel** gibt es noch weitere Bezeichnungen, die aus der biologischen Analogie entlehnt wurden, z.B. das **Blatt**. So wird ein Knoten genannt, der keinen Nachfolger mehr besitzt, d.h. im Falle des binären Baumes, dass die Referenzen `left` und `right` den Wert

NULL besitzen müssen.

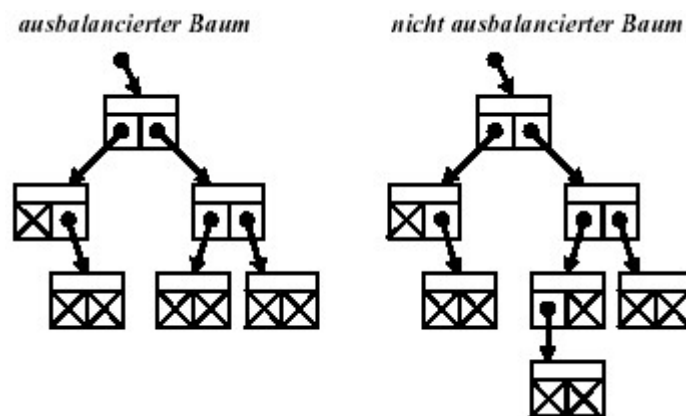
So wie die Darstellungen der linearen Listen kann auch die des binären Baumes als gerichteter Graph gelesen werden. **Ast** nennt man dann einen Weg vom Wurzelknoten bis zu einem Blatt. Zu beachten ist dabei, dass beim binären Baum der Weg von der Wurzel bis hin zu einem bestimmten Blatt immer eindeutig ist. Die Länge eines Astes wird definiert durch die Anzahl der Knoten, die dieser Weg beinhaltet.

Als **Tiefe** eines Baumes wird das Maximum aller Astlängen bezeichnet. Mit anderen Worten, die Tiefe ist die **Anzahl der Ebenen**, die ein Baum in seiner momentanen Gestalt belegt.

Weiters ist der Begriff des **ausbalancierten** Baumes wichtig. So bezeichnet man einen Baum, der nicht mehr Ebenen als unbedingt notwendig besitzt für eine bestimmte Anzahl (N) an gespeicherten Datenelementen:

Minimum der benötigten Ebenen = $\lceil \lg(N+1) \rceil$ , bzw. 0 für $N = 0$
---

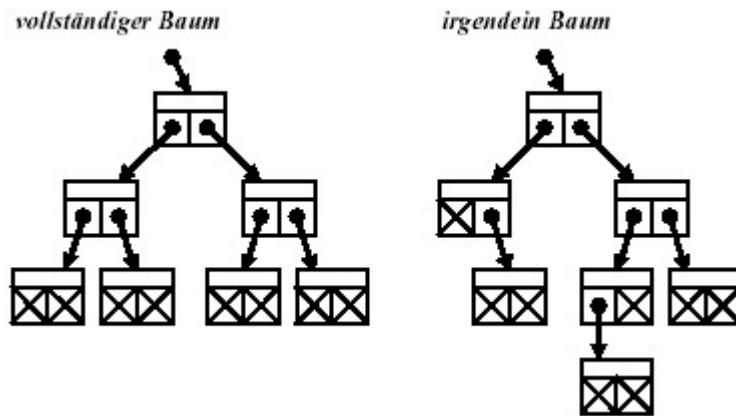
In der Grafik gilt der linke Baum als ausbalanciert, da für 6 Knoten beim binären Baum nun mal mindestens drei Ebenen benötigt werden. Beim rechten hingegen ist die Verwendung der vierten Ebene nicht unbedingt notwendig, da 7 Knoten auch in drei Ebenen Platz hätten. In dem hier gezeigten Fall wäre eine Umsortierung der Knoten notwendig, um dieses Ziel zu erreichen, da nicht einfach der Knoten in der vierten Ebene in den freien Platz verschoben werden kann (das darin gespeicherte Datenelement ist größer bzw. gleich groß als das im Wurzelknoten, darf daher nicht links davon zu liegen kommen!).



Ein Baum wird als vollständig bezeichnet, wenn es sich um einen ausbalancierten Baum handelt, dessen letzte Ebene voll besetzt ist. Vollständige Bäume müssen daher eine bestimmte Anzahl an Datenelementen enthalten (hier der Fall des binären Baumes):

notwendige (nicht hinreichende) Bedingung: $N = 2^x - 1$ ,
--

d.h. die Anzahl der Knoten muss eine Zweierpotenz minus 1 sein. In der folgenden Grafik erfüllen beide Bäume die angegebene Formel, trotzdem ist nur der linke als vollständig zu bezeichnen, da dieser zusätzlich noch ausbalanciert ist:



Und zu guter letzt noch eine Formel für binärer Baume, deren Herleitung wohl niemandem Kopfzerbrechen bereiten würde:

$\text{Maximale Anzahl d. Knoten einer Ebene} = 2^{\text{Ebene}}$
---

### 15.3.2 Anwendungsgebiete

Es wurde bereits angesprochen, Bäume sind, ganz allgemein, ideale Datenstrukturen zum Suchen gespeicherter Datenelemente bzw. zur Klärung der Frage, ob das Element der Begierde überhaupt vorhanden ist.

Doch bleiben wir beim binären Baum. Hier ist der Aufbau und die damit verbundene Sortierung der Daten für diese Fähigkeit verantwortlich. Denn bei einem bestehenden Baum ist der Weg von der Wurzel zu dem gesuchten Datenelement (egal, ob vorhanden oder nicht) immer eindeutig. Das bedeutet wiederum, dass die Anzahl der Suchschritte (Anzahl der besuchten Knoten) durch die Größe des längsten Astes (d.h. der Tiefe des Baumes) nach oben hin beschränkt ist.

Die folgende Tabelle listet auf, wie **viele Suchschritte** maximal (im optimalen Fall) notwendig sind, um ein gespeichertes Datenelement zu finden oder festzustellen, dass dieses nicht vorhanden ist. Diese Zahlen errechnen sich aus der bereits vorgestellten Formel für die minimale Anzahl an benötigten Ebenen für eine gegebene Menge an Datenelementen:

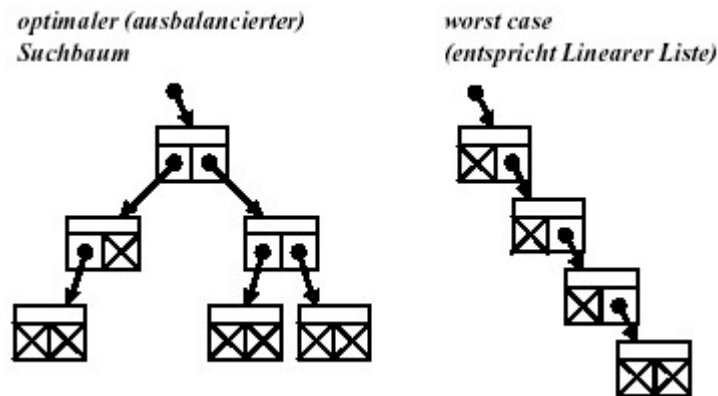
Anzahl der Datenelemente (N)	Anzahl notwendiger Suchschritte (optimal)
1	1
10	4
100	7
1.000	10
10.000	14
100.000	17
1.000.000	20

Das alles gilt aber nur für den Idealfall, dass nämlich der binäre Baum ausbalanciert ist. Denn nur in diesem Fall ist die Tiefe des Baumes für gegebene Anzahl an gespeicherten Datenelementen minimal. Ist der Baum nicht ausbalanciert, so existiert mindestens ein Knoten, der erst mit mehr Suchschritten als notwendig erreichbar ist.

Normalerweise hat man aber nicht das Glück, dass ein Baum nach Einfügen der verschiedenen Datenelemente ausbalanciert vorliegt. Der schlimmste mögliche Fall („worst case“) kommt dann zustande, wenn die Daten, die in den Binären Baum eingefügt werden

sollen, bereits sortiert vorliegen und dies nicht besonders behandelt wird. Denn was passiert in diesem Fall?

Man nimmt das erste Datenelement und speichert es im neu angelegten Wurzelknoten. Geht man davon aus, dass die Daten aufsteigend sortiert sind (für absteigend sortierte Daten gilt analoges), so ist das zweite Element größer als das erste und muss demnach rechts vom Wurzelknoten gespeichert werden. Das nun folgende dritte Element ist größer als beide Vorgänger und wird daher im rechten Unterbaum des zweiten Knoten abgelegt und so geht es weiter. Was bei diesem Vorgang herauskommt, ist eine sortierte lineare Liste, bei der nur die Namen verschiedener Referenzen ausgetauscht wurden (root statt listBegin und right statt next):



Ist der Baum wirklich in einer linearen Liste ausgeartet, dann müssen im ungünstigsten Fall alle Datenelemente überprüft werden, bis das gesuchte Element gefunden wird (bzw. es feststeht, dass es nicht gespeichert ist). Und auch wenn man nicht so pessimistisch ist, im Mittel muss immer noch die Hälfte der Liste durchforstet werden, um diese Entscheidungen treffen zu können. Schon bei hundert Datensätzen ein unglaublicher Mehraufwand gegenüber einem ausbalancierten Baum (siehe Tabelle).

### 15.3.3 Beispiel

Die einzelnen Funktionen zum Erstellen eines binären Baumes sind ohne Verwendung einer rekursiven Lösung schwer zu implementieren.

Bei der Funktion deleteElement() ist eine Fülle von Spezialfällen zu berücksichtigen, die im Folgenden kurz aufgezählt werden:

- leerer Baum
- ein Gefundenes Element kann als gelöscht markiert sein
- zu löschendes Element im Wurzelknoten des Gesamtbaumes (Referenz root direkt betroffen):
- es existiert kein rechter Unterbaum
- Element gleich rechts vom zu löschenden Knoten soll neues Wurzelement werden (ist demnach das kleinste Datenelement im rechten Unterbaum)
- neues Wurzelement ganz links im rechten Unterbaum (Behandlung dieses Falls: siehe folgendes Programmstück)
- zu löschendes Element nicht im Wurzelknoten des Gesamtbaumes:
- ein Blatt wird gelöscht (einfachster Fall)
- es existiert kein rechter Unterbaum
- ... (wie oben)

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

```

#include <conio.h>
#include <windows.h>

typedef struct TREEELEMENT{
    int data;
    struct TREEELEMENT *left;
    struct TREEELEMENT *right;
}TreeElement;

int insertElement(int data, TreeElement **root);
void insertSubTree(int data, TreeElement *tel);
TreeElement *findElement(int data, TreeElement *tel);
int deleteElement(int data, TreeElement **root);
void dataOutput(TreeElement *tel);

int main(int argc, char *argv[]){
    TreeElement *treeBegin;

    treeBegin=NULL;

    insertElement(2, &treeBegin);
    insertElement(1, &treeBegin);
    insertElement(5, &treeBegin);
    deleteElement(2, &treeBegin);
    findElement(1, treeBegin);
    dataOutput (treeBegin);
    return EXIT_SUCCESS;
}

int insertElement(int data, TreeElement **root){
    TreeElement *te=NULL;

    /* Spezialfall: Baum ist leer */
    if (*root==NULL){
        if ((te=(TreeElement *)malloc(sizeof(TreeElement)))==NULL){
            fprintf(stderr,"Nicht genugend Speicher verfuegbar!");
            return EXIT_FAILURE;
        }
        *root=te;
        te->data=data;
        te->left=NULL;
        te->right=NULL;
        return EXIT_SUCCESS;
    }

    insertSubTree(data, *root);
    return EXIT_SUCCESS;
}

void insertSubTree(int data, TreeElement *tel){
    TreeElement *te=NULL;

    /*Datum gehört in den linken Zweig*/
    if (data<(tel->data)){
        if ((tel->left)==NULL){
            if ((te=(TreeElement *)malloc(sizeof(TreeElement)))==NULL){
                fprintf(stderr,"Nicht genugend Speicher verfuegbar!");
                exit(EXIT_FAILURE);
            }
            te->left=NULL;
            te->right=NULL;
            te->data=data;
            tel->left=te;
        }
        else{
            insertSubTree(data,tel->left);
        }
    }

    /*Datum gehört in den rechten Zweig*/
    else{
        if ((tel->right)==NULL){

```



```

        if ((te=(TreeElement *)malloc(sizeof(TreeElement)))==NULL) {
            fprintf(stderr, "Nicht genug Speicher verfuegbar!");
            exit(EXIT_FAILURE);
        }
        te->left=NULL;
        te->right=NULL;
        te->data=data;
        tel->right=te;
    }
    else{
        insertSubTree(data, tel->right);
    }
}

TreeElement *findElement(int data, TreeElement *tel){

    if (tel==NULL){
        //printf("Nicht gefunden\n");
        return NULL;
    }
    if (data<tel->data){
        return findElement(data, tel->left);
    }
    else if (data>tel->data){
        return findElement(data, tel->right);
    }
    else {
        //printf("Gefunden: %d\n", tel->data);
        return tel;
    }
}

int deleteElement(int data, TreeElement **root){

    /*todo: Daten des Elements, das gerade gelöscht worden ist */

    TreeElement *prev=NULL, *tel=*root;

    if (tel==NULL) return EXIT_FAILURE;

    /* ist das Datum gefunden worden? */
    if ((*root)->data==data){

        /* ein Blatt, das gelöscht werden soll ?*/
        if (((*root)->left==NULL)&&((*root)->right==NULL)){
            *root=NULL;
            return EXIT_SUCCESS;
        }

        /* ein Knoten, der nur mehr einen linken Unterbaum hat, soll gelöscht werden */
        else if (((*root)->right==NULL)&&((*root)->left)){
            (*root)=(*root)->left;
            return EXIT_SUCCESS;
        }

        /* ein Knoten, der nur mehr einen rechten Unterbaum hat, soll gelöscht werden
    */
        else if (((*root)->left==NULL)&&((*root)->right)){
            (*root)=(*root)->right;
            return EXIT_SUCCESS;
        }

        /* "Normalfall" */
        else{
            tel=(*root);
            prev=tel;
            tel=tel->right;

            while(tel->left){
                prev=tel;
                tel=tel->left;
            }

```

```

    }
    (*root)->data=tel->data;

    /*Nur mehr ein Element nach dem zu löschenden Knoten?*/
    if (((*root)->right->left)==NULL && ((*root)->right->right==NULL)){
        (*root)->right=NULL;
        free(tel);
        return EXIT_SUCCESS;
    }
    if (prev==(*root)){
        (*root)->right=tel->right;
        free(tel);
        return EXIT_SUCCESS;
    }
    /* nach dem zu löschenden Knoten folgt ein Element, das noch einen
    rechten Unterbaum hat (oder auch nicht) */
    prev->left=tel->right;
    free(tel);
    return EXIT_SUCCESS;
}

}
else if (data<(*root)->data){
    return deleteElement(data,&((*root)->left));
}
else{
    return deleteElement(data,&((*root)->right));
}
}

/* Spezialfall: nur ein Element im Baum vorhanden */
if ((((*root)->left)==NULL)&& ((*root)->right)==NULL)&& ((*root)->data==data)){
}

if ((temp=findElement(data,tel))==NULL){
    return EXIT_FAILURE;
}

/* Spezialfall: Blatt löschen
if ((temp->left==NULL)&&(temp->right==NULL)){
    prev=temp;
    free(prev);
    temp=NULL;
    return EXIT_SUCCESS;
}

/*Spezialfall: nur linker Zweig vorhanden
if (((temp->right)==NULL)&&(temp->left)){
    tel=temp->left;
    temp->data=tel->data;
    temp->left=tel->left;
    temp->right=tel->right;
    free(tel);
    return EXIT_SUCCESS;
}

/*Normalfall
tel=temp;
prev=tel;
tel=tel->right;
//printf("da bin ich %d",tel->data);
while(tel->left){
    prev=tel;
    tel=tel->left;
}
prev->left=NULL;
temp->data=tel->data;
free(tel);

return EXIT_SUCCESS;
}

*/
void dataOutput(TreeElement *tel){

```

```
    if (tel->left){  
        dataOutput(tel->left);  
    }  
    printf("%d\n",tel->data);  
    if (tel->right){  
        dataOutput(tel->right);  
    }  
}
```

## 15.4 Spezielle Datenstrukturen

Für Hashing und AVL-Baum ist die ausreichende Zeit zu kurz, für Interessierte sei auf die Literatur verwiesen bzw. bitte ich Sie, sich an den Vortragenden zu wenden.

## 16 Der Präprozessor

Bevor der eigentliche Compiler an die Übersetzung eines C/C++ -Quelltextes geht, wird dieser zunächst von dem C/C++ -Präprozessor bearbeitet. Der C/C++ -Präprozessor ist ein einfacher Makro Prozessor, mit dem sich Spracherweiterungen implementieren lassen. In den bislang geschriebenen Programmen haben wir von einer Funktionalität des C/C++ -Präprozessors schon oft Gebrauch gemacht, nämlich von der des Einfügens eines Quelltextes in einen anderen durch die `#include`-Anweisung. Präprozessoranweisungen beginnen immer mit dem Doppelkreuz (`#`). Dieses Doppelkreuz muss immer das erste Zeichen einer Zeile sein, wobei optional führende Leerzeichen erlaubt sind. Der C/C++ -Präprozessor kennt Direktiven

- zum Einfügen eines Quelltextes in einen anderen
- zum Textersatz
- und zur bedingten Übersetzung

### 16.1 Textersatz

Der Präprozessor kann einfache Textersetzungen in C/C++ -Quellen vornehmen. Der Textersatz in Programmen basiert auf der `#define`-Direktive. Mit dieser Direktive lassen sich so genannte Präprozessor - Makros (oder kurz Makros) definieren. Tritt ein solcher Makro dann später in einer C/C++ -Quelle auf, so wird er durch seine Definition ersetzt. Wird etwa der Makro `ZEILENLAENGE` durch die Präprozessor Direktive

```
#define ZEILENLAENGE 80
```

definiert, so wird im folgenden C/C++ -Quelltext jedes Auftreten des Wortes `ZEILENLAENGE` außerhalb von Zeichenkettenkonstanten durch `80` ersetzt. Die Deklaration eines `char` - Vektors zur Aufnahme einer Textzeile kann dann also durch

```
char textzeile [ ZEILENLAENGE ];
```

erfolgen.

Darüberhinaus gestattet der C/C++ -Präprozessor es, parameterisierte Makros zu definieren. Ein Beispiel eines parameterisierten Makros zur Berechnung des Maximums zweier Werte `A` und `B` wäre etwa

```
#define MAX(A,B) ((A) > (B) ? A : B)
```

Nach der Definition des Präprozessormakros `MAX` wird nun jedes Auftreten der Zeichenfolge

```
MAX(Argument1,Argument2)
```

in einem C/C++ -Programm durch die Zeichenfolge

```
((Argument1) > (Argument2) ? Argument1 : Argument2)
```

ersetzt. Bei der Definition und Verwendung von parameterisierten Präprozessormakros ist darauf zu achten, dass sehr leicht unerwünschte Effekte entstehen können, wie das folgende Beispiel zeigt:

```
ergebnis = MAX ( x++, y++ );
```

Die Textersetzung hierbei ergibt:

```
ergebnis = ((x++) > (y++) ? x++ : y++);
```

Man erkennt leicht, dass, wenn x größer als y ist, die Variable x zweimal inkrementiert wird. Im anderen Fall wird die Variable y doppelt inkrementiert. Dieser Effekt ist vom Programmierer sicherlich nicht beabsichtigt!

Des Weiteren ist es immer sinnvoll, die Parameter von Präprozessormakros einzuklammern, wie das folgende Beispiel eines unsauber geklammerten Präprozessormakros zeigt:

```
#define MULT(X,Y) X*Y
```

Die Verwendung des Makros MULT könnte dann etwa so aussehen:

```
z = MULT( 2 + 3, 4 + 5 );
```

Nach der Expansion des Makros, also nach dem Textersatz ergibt sich:

```
z = 2 + 3 * 4 + 5;
```

Das Ergebnis ist 19 und nicht, wie vom Programmierer sicherlich erwartet 45. Dies liegt daran, dass nach der Expansion des Makros die Multiplikation  $3 * 4$  einen höheren Vorrang hat als die beiden Additionen! Ein sauber definierter Präprozessormakro MULT sollte also etwa so aussehen:

```
#define MULT(X,Y) ((X)*(Y))
```

Jetzt ergibt sich nach der Expansion der Ausdruck

```
z = ((2 + 3) * (4 + 5));
```

Dieser ergibt nach der Bewertung dann das erwartete Ergebnis.

### Aufhebung von Makrodefinitionen

Die Definition eines mittels der #define-Direktive vereinbarten Makros kann durch die

```
#undef <Makroname>
```

Direktive aufgehoben werden.

## 16.2 Bedingte Übersetzung

Der C/C++ -Präprozessor kann auch dazu benutzt werden, abhängig von bestimmten Bedingungen Teile des Quelltextes von der Übersetzung durch den C/C++ -Compiler auszunehmen oder gezielt einzuschließen. Der C/C++ -Präprozessor kennt die folgenden Direktiven zum Testen von Bedingungen:

**#if <Ausdruck>:** Der hiervon abhängige Quelltext wird übersetzt, falls <Ausdruck> von 0 verschieden ist.

`#ifdef (<Makroname>)` : Der hiervon abhängige Quelltext wird übersetzt, falls der Präprozessormakro `<Makroname>` definiert ist.

`#ifndef (<Makroname>)` : Der hiervon abhängige Quelltext wird übersetzt, falls der Präprozessormakro `<Makroname>` nicht definiert ist.

Diese Direktiven können durch eine der folgenden beiden Direktiven fortgesetzt werden:

`#else`: Der hiervon abhängige Quelltext wird übersetzt, falls der vom zugeordneten `#if...` abhängige Quelltext nicht übersetzt wird.

`#elif <Ausdruck>` : Der hiervon abhängige Quelltext wird übersetzt, falls `<Ausdruck>` von 0 verschieden ist.

Den Abschluss bildet in jedem Fall die Direktive `#endif`. Die in den Präprozessordirektiven zur bedingten Übersetzung verwendeten Ausdrücke werden aus den üblichen logischen, arithmetischen und Vergleichsoperatoren (+, -, \*, /, ..., ==, !=, ..., &&, ||, !, ...) sowie dem Operator `defined (<Makroname>)` gebildet. Der Operator `defined (<Makroname>)` liefert genau dann den Wahrheitswert wahr, wenn der Präprozessormakro `<Makroname>` definiert ist, und falsch sonst. Demzufolge sind die folgenden Präprozessordirektiven äquivalent:

```
#ifdef <Makroname>
```

```
#if defined (<Makroname>)
```

beziehungswiese

```
#ifndef <Makroname>
```

```
#if !defined (<Makroname>)
```

Alle Präprozessordirektiven zur bedingten Übersetzung dürfen ineinander geschachtelt werden. Die Abbildung 9.1 zeigt ein kleines Beispiel zur Verwendung der bedingten Übersetzung.

```
1 #include <stdio.h>
2
3 #define GERMAN 1
4 #define ENGLISH 2
5 #define FRENCH 3
6
7 #define LANGUAGE GERMAN
8
9 int
10 main ()
11 {
12     #if LANGUAGE == GERMAN
13         printf ("hallo welt\n");
14     #elif LANGUAGE == ENGLISH
15         printf ("hello world\n");
16     #else
17         printf ("bonjour monde\n");
```

```
18 #endif
19 }
```

## 17 Programmbausteine im Quellcode

Die einfachste Form Programmbausteine zu verwenden sind getrennte Dateien für jeden schon fertigen und erprobten Programmteil. Es kann beispielsweise eine Funktion oder eine Gruppe von Funktionen als Quellcode, das heißt als C-Programm oder Programmteil, in eine eigene Datei geschrieben werden.

Dieser Baustein kann dann in jedes einzelne Programm, wo er benötigt wird, hineinkopiert werden.

Wesentlichster Nachteil: wenn der Baustein nachträglich geändert werden muss, muss der (neue) Baustein neuerlich in alle Programme, in denen er benötigt wird, hineinkopiert werden. Dabei kann, vor allem bei sehr großen Projekten, leicht ein Programm übersehen werden.

Etwas besser sind "include"-Dateien. Diese Bausteine wurden bisher bereits ab dem ersten Beispiel verwendet: die Prototypen der verwendeten Unterprogramme und sonstige Vereinbarungen werden bei jeder Übersetzung neuerlich gelesen. Eine include - Datei kann auch für selbst geschriebene Programme verwendet werden.

In das Programm wird der Name der Datei geschrieben, die den Baustein enthält.

In jedem Programm steht dann (an passender Stelle) die Zeile `#include <irgendwas.c>` oder `"irgendwas.c"`. Wird die zweite Form verwendet, sucht der Compiler die Datei im aktuellen Verzeichnis.

Include-Dateien sind einfach zu handhaben, bringen aber, wenn sie umfangreich werden, Nachteile: wenn im Hauptprogramm (das sehr kurz sein kann) auch nur eine Zeile geändert wird, müssen alle Programmteile, auch alle include-Dateien, wieder übersetzt werden.

### 17.1 Übergabe der Steuerung an andere Hauptprogramme

Eine weitere Möglichkeit, umfangreichere Aufgaben übersichtlich zu bearbeiten, besteht darin, dass ein Hauptprogramm nach Ende seiner Arbeit ein anderes Hauptprogramm aufruft. Dieses Vorgang, oft als "chaining" ("chain" = "Kette") bezeichnet, hat den Vorteil, dass jedes Hauptprogramm für sich entwickelt und erprobt werden kann. Erst am Ende der gesamten Entwicklung wird aus allen Hauptprogrammen ein komplettes Softwarepaket gemacht.

```
spawnl (m,name,arg0,arg1,...,NULL)
```

dient zum Verketteten von Programmen. Das aufgerufene Programm heißt auch Tochterprozess. Von einem Hauptprogramm aus kann ein beliebiges anderes Programm mit der Prozedur `spawnl` aufgerufen werden. `name` ist eine Zeichenkette, die den vollständigen Dateinamen des aufzurufenden Programms enthält. `arg0`, `arg1` usw. sind beliebig viele (auch 0) Parameter, die an das gerufene Programm übergeben werden. Als Abschluss dieser Parameterliste, das heißt als letzter Parameter von `spawnl`, muss `NUL` stehen.

Der Parameter `m` bestimmt, was nach Ende des aufgerufenen Programms geschehen soll. Folgende Konstanten sind in `process.h` vordefiniert:

`P_WAIT` veranlasst das rufende Programm auf das Ende des gerufenen Programms zu warten.

`P_NOWAIT` sollte rufendes und gerufenes Programm gleichzeitig weiterlaufen lassen

`P_OVERLAY` bewirkt, dass das gerufene Programm das rufende im Speicher ersetzt und verdrängt.



Liefert `spawnl` den Funktionswert -1, ist ein Fehler aufgetreten.

Beispiel:

```
int res= 0;
res=spawnl(P_WAIT, "C:\\X.EXE", "C:\\X.EXE", NULL);
if (res==-1) fprintf(stderr, "Fehler in spawnl");
```

Nachdem das gerufene Programm fertig ist, nimmt in diesem Beispiel das rufende Programm seine Arbeit wieder auf.

Mit dem Aufruf von `execl(name, arg0, arg1, ..., NULL)` kann auch ein weiteres Hauptprogramm gestartet werden. Die Parameter haben dieselbe Bedeutung wie in `spawnl`. Eine Rückkehr in das rufende Programm ist nicht möglich.

Zur Vollständigkeit, sei auch noch der Befehl `system("C:\\X.EXE")` erwähnt, welcher wie bei `P_WAIT` auf das Ende des aufgerufenen Programms wartet.

## 17.2 Objekt-Dateien

Programmbausteine sind meistens Unterprogramme. Wenn diese Unterprogramme erprobt sind, werden sie (einmal) übersetzt und in einer Bibliothek abgelegt. Übrigens müssen diese Unterprogramme nicht unbedingt in C geschrieben sein - über Programmbibliotheken können Unterprogramme verschiedener Sprachen verbunden werden.

In der Bibliothek stehen die Programmbausteine allen Benutzern zur Verfügung. Bei Übersetzen des Hauptprogramms brauchen die schon fertigen Bausteine nicht mehr mitübersetzt zu werden. Übersetztes Hauptprogramm und vorübersetzte Unterprogramme werden erst nachträglich vom "Binder" ("Linker") zu einem lauffähigen Programm vereinigt.

**Im Einzelnen sind folgende Schritte notwendig:**

1. Zuerst muss festgelegt werden, auf welche Unterprogramme und auf welche sonstigen Informationen der Benutzer der Bibliothek zugreifen darf. Dabei werden die Schnittstellen festlegt:

Namen der Unterprogramme, ferner Anzahl, Reihenfolge, Typ und Bedeutung der Parameter.

2. Dann wird bestimmt, welche Aufgabe die einzelnen Unterprogramme im Detail haben sollen: der Ablauf in den Unterprogramme kann beispielsweise durch Struktogramme beschrieben werden. Ferner können Unterprogramme auch nur als Hilfsprogramme verwendet werden, das heißt nicht dem Benutzer zugänglich sein.

3. Die Unterprogramme des zweiten Schrittes werden in C (oder auch einer anderen Sprache) programmiert und separat übersetzt.

4. Nun zu den Programmen, die die erzeugten Bibliotheksprogramme verwenden wollen: In jedem Hauptprogramm müssen zur syntaktisch richtigen Übersetzung die Informationen des ersten Schrittes sichtbar gemacht werden.

5. Wenn das Hauptprogramm übersetzt ist, müssen das Hauptprogramm und die vorübersetzten Unterprogramme miteinander verbunden werden.

Der Vollständigkeit halber sei noch erwähnt, dass die Unterprogramme des ersten Schrittes statt in einem Hauptprogramm auch in anderen Unterprogrammen verwendet werden können.

## 17.3 Das Entwickeln und Warten von Programmen (make) unter LINUX (UNIX)

Es gibt in UNIX ein sehr mächtiges Werkzeug zum Entwickeln und Warten von Programmen, das Kommando `make`. Dieses Kommando liest in einer Datei, die standardmäßig `Makefile` heißt, Informationen darüber, wie aus Sourcefiles ein ausführbares Programm erzeugt werden kann. Diese Information wird dazu benutzt, um alle Dateien (Object-Files, ausführbares Programm, ...), die automatisch erstellt werden können und die nicht mehr up-to-date sind, neu zu generieren (compilieren, linken).

### Dazu ein Beispiel:

Ein Programm besteht aus den Modulen `x.c`, `y.c` und `z.c` mit den entsprechenden Header - Dateien `x.h`, `y.h` und `z.h`. Dabei wird jede Header-Datei vom entsprechenden C-File inkludiert, und außerdem wird `y.h` von `x.c` und `z.h` von `y.c` inkludiert. Das exekutierbare Programm heißt `xyz`.

Wird nun die Datei `y.c` geändert, so muss das Object-File zu `y.c` (`y.o`) neu erzeugt werden, und ebenso das exekutierbare Programm `xyz` aus den Object-Files `x.o`, `y.o` und `z.o`.

Wird z.B. die Datei `y.h` geändert, so müssen alle Module, die diese Datei inkludieren (`y.c` und `x.c`) neu compiliert werden und außerdem müssen die Object-Files neu gelinkt werden.

*Wie sieht nun ein Makefile aus, das ein solches Programm beschreibt?*

Ein Makefile besteht aus einer Liste von Einträgen, die beschreiben, wie jeweils eine Datei neu erzeugt werden kann. Außerdem wird in jedem Eintrag noch angegeben, nach der Änderung welcher Dateien die angegebene Datei neu erzeugt werden muss.

Ein Eintrag sieht folgendermaßen aus:

```
Target: Dependencies
```

```
<tab> Kommando
```

Target ist entweder der Name einer Datei, die durch die folgenden Kommandos neu erzeugt wird, oder es ist ein Label.

Dependencies ist eine Liste, die Dateinamen und Namen von Targets enthält. Diese Liste hat folgende Bedeutung: Zuerst werden die Regeln für alle Targets, die in der Liste der Dependencies enthalten sind, ausgeführt. Anschließend wird überprüft, ob mindestens eine der in der Liste der Dependencies angeführten Dateien (oder Targets) geändert wurde, nachdem Target das letzte Mal erzeugt wurde. Ist dies der Fall, so werden alle Kommandos, die sich nach der Zeile mit dem Target befinden, ausgeführt.

ACHTUNG! Jedes dieser Kommandos muss mit einem Tabulator beginnen! Die erste Zeile nach Target, die nicht mit einem Tabulator beginnt, gibt das Ende der Liste der Kommandos an.

Ist ein Target kein Dateiname, sondern ein Label, so wird genauso vorgegangen, mit dem Unterschied, dass die Kommandos, die dem Target folgen, auf jeden Fall ausgeführt werden. Die Verwendung von Labels als Targets dient vor allem dazu, um mit dem `make` Kommando verschiedene Aktionen, die im Makefile beschrieben sind, auszuführen.

Wird das Kommando make ohne Parameter aufgerufen, so wird versucht, das erste Target im entsprechenden Makefile zu bilden. Es kann aber auch ein Target als Parameter für make angegeben werden. In diesem Fall wird versucht, dieses Target zu bilden.

Es ist Konvention, dass folgende zwei Labels in einem Makefile angegeben sind:

**all:** Dieses Label dient dazu, um das ganze Programm (bzw. Programmpaket) neu zu erstellen. Als Dependencies von all sollen alle exekutierbaren Dateien angegeben werden, die zu dem Programmpaket gehören. all: muss immer das erste Target in einem Makefile sein, damit bei Eingabe von make ohne Parameter immer das gesamte Programmpaket neu erstellt wird.

**clean:** Dieses Label dient dazu, um alle Dateien, die aus irgendeiner anderen Datei automatisch erzeugt werden können, zu löschen. Dies sind normalerweise alle Object-Files und die exekutierbaren Programme.

Das Makefile, das das oben angeführte Programm beschreibt, sieht wie folgt aus:

```
##
## Project: xyz
## Module:  Makefile
## File:    Makefile
## Version: 1.1
## Contents:      Makefile for project xyz

all: xyz

xyz: x.o y.o z.o
    gcc -o xyz x.o y.o z.o

x.o: x.c x.h y.h
    gcc -ansi -Wall -g -c x.c

y.o: y.c y.h z.h
    gcc -ansi -Wall -g -c y.c

z.o: z.c z.h
    gcc -ansi -Wall -g -c z.c

clean:
    rm -f x.o y.o z.o xyz
```

Noch zwei Anmerkungen zu Makefiles:

- Ist eine Zeile länger als die Bildschirmbreite, so kann man sie auf mehrere Zeilen aufteilen, wobei jede Zeile bis auf die letzte mit einem Backslash (\) enden muss.
- Das Kommentarzeichen in Makefiles ist '#'.