Programmieren I

Computertechnik
Datentypen
Operatoren
Ausdrücke
Kontrollstrukturen
Funktionen
Vektoren
Programmstruktur

HAW-Hamburg
Fakultät Technik und Informatik
Department Informations- und Elektrotechnik

Robert Heß

18. März 2019

Inhaltsverzeichnis

Kapitel 1

Überblick Computertechnik

1.1 Interner Aufbau eines Computers

Betrachten wir ein typisches Computersystem von außen, so entdecken wir eine Tastatur, eine Maus oder einen Trackball, einen Bildschirm, vielleicht auch einen Drucker und einen Scanner. Alle diese Geräte sind zentral mit dem eigentlichen Computer verbunden. Beim näheren Betrachten entdecken wir im Computer selbst noch weitere Geräte zum Lesen und Schreiben von und CDs und DVDs etc., siehe Abbildung 1.1.

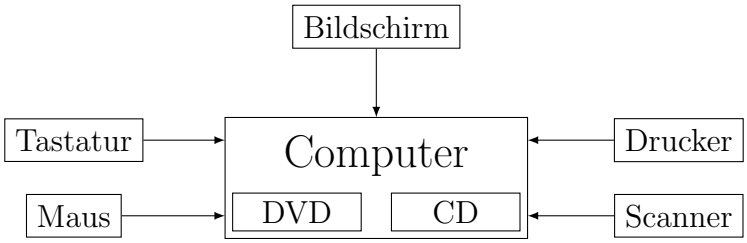


Abbildung 1.1: Äußere Elemente eines typischen Computersystems.

Schon beim Kauf eines Computers merkt man aber schnell, dass sich der Computer selbst auch aus mehreren Teilen zusammensetzt. Da geht es um Speicher/RAM, Festplatten, Taktfrequenzen etc. Abbildung 1.2 zeigt vereinfacht das Innenleben eines Computers.

Die *Central Processing Unit* (CPU) ist das Herzstück eines jeden Computers. In ihr findet die Verarbeitung aller Daten statt. Sei es eine Mausbewegung, eine Eingabe über die Tastatur, Änderungen auf dem Bildschirm: immer ist die CPU daran beteiligt. Damit die CPU mit der Außenwelt kommunizieren kann, ist sie mit einem Datenbus, einem Adressbus und einem Steuerbus versehen. Hinter einem Bus verbergen sich eine Anzahl von elektrischen Leitungen, die jeweils nur zwei Zustände einnehmen können. Die Breite der Busse (Anzahl der el. Kontakte) variiert. Typische Breiten für Adress- und Datenbus sind 32 Leitungen. Der Steuerbus umfasst ca. 10 oder mehr Anschlüsse.

Über den Datenbus erfolgt der Transport von Informationen, den Daten. Daten können entweder von der CPU gelesen, also zur CPU hin transportiert werden, oder Daten können von der CPU geschrieben, also von der CPU weg transportiert werden. Der Datenbus ist somit bidirektional.

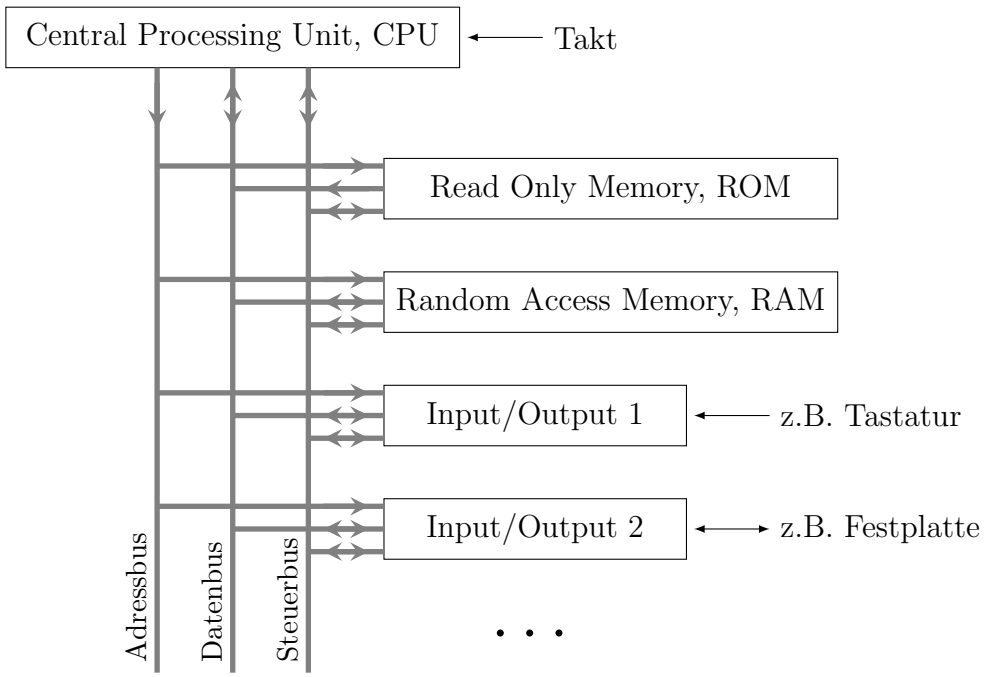


Abbildung 1.2: Vereinfachter interner Aufbau eines Computers.

Die zur Verfügung stehende Datenmenge ist deutlich zu groß, als dass sie parallel mit der CPU verdrahtet werden könnte. Zu einem Zeitpunkt kann nur ein kleiner Teil von Daten übertragen werden. Die Selektion der Daten erfolgt über den Adressbus. Der Adressbus wird von der CPU ausschließlich zur Selektion der Daten verwendet und ist somit unidirektional. Die Arbeitsweise des Adressbusses wird im Abschnitt 1.2 erläutert.

Der Steuerbus enthält Leitungen mit unterschiedlichen Funktionen. Die Belegung dieser Leitungen ist für verschiedene Systeme recht unterschiedlich. Typischerweise gibt eine Leitung an, ob die CPU gerade Daten transferieren möchte oder nicht. Eine andere Leitung gibt an, ob Daten gelesen oder geschrieben werden. Eine weitere Leitung wird aktiviert, wenn eine Schnittstelle (I/O) angesprochen werden soll. Mit einigen Leitungen im Steuerbus lässt sich die CPU unterbrechen (Interrupt). Der Steuerbus ist bidirektional: Einige Leitungen senden Signale, andere empfangen Signale, und es gibt Leitungen, die Signale senden und empfangen.

Die Externen Geräte werden über Schnittstellen, den I/Os (Input/Output) angesprochen. Z.B. wird beim Betätigen der Taste 'A' auf der Tastatur über den Steuerbus der CPU mitgeteilt, dass eine neue Information vorliegt, die dann von der CPU gelesen wird. Auf die recht komplexe Arbeitsweise der Schnittstellen wird hier nicht weiter eingegangen.

1.2 Funktionsweise des Speichers

Ein heutiger Computer hat üblicherweise einen Arbeitsspeicher von einem Gigabyte oder mehr. Das sind $2^{30} = 1.073.741.824$ Byte, wobei jedes Byte aus acht Speicherzellen besteht, was $2^{33} = 8.589.934.592$ Speicherzellen entspricht. (In der Computertechnik steht die Vorsilbe *Kilo*, angedeutet durch ein großes K, nicht für 10^3, sondern für 2^{10}, ebenso *Mega* für 2^{20} und *Giga* für 2^{30}.)

In fast allen Computern sind die Speicher Byte-weise, also zu Blöcken von acht Speicherzellen organisiert. Jedes Byte erhält eine eindeutige (Haustür-) Nummer, die Adresse (siehe Abbildung 1.3). Damit kann jede Speicherzelle eindeutig identifiziert werden. Auch wenn ein heutiger Computer gleich vier oder acht Byte auf einmal einliest, hat immer noch jedes Byte seine eigene Adresse.

Adresse:		Datum:
0	→	01100101
1	→	11100101
2	→	11111000
3	→	10100101
4	→	11010111
5	→	00001010
6	→	00110010

Abbildung 1.3: Adressen und (willkürlicher) Inhalt von Speicherzellen.

Will nun die CPU auf eine bestimmte Speicherzelle zugreifen, so stellt sie auf dem Adressbus die gewünschte Adresse ein. Im Speicherblock wird die Adresse dekodiert, und das entsprechende Byte bekommt ein Freigabezeichen, siehe Abbildung 1.4. Nun teilt die CPU dem Speicher noch über den Steuerbus mit, ob es in die Speicheradresse schreiben will, oder dessen Inhalt auslesen möchte. Danach ist die Verbindung zwischen CPU und der gewünschten Stelle im Speicher hergestellt, und die Daten werden übertragen.

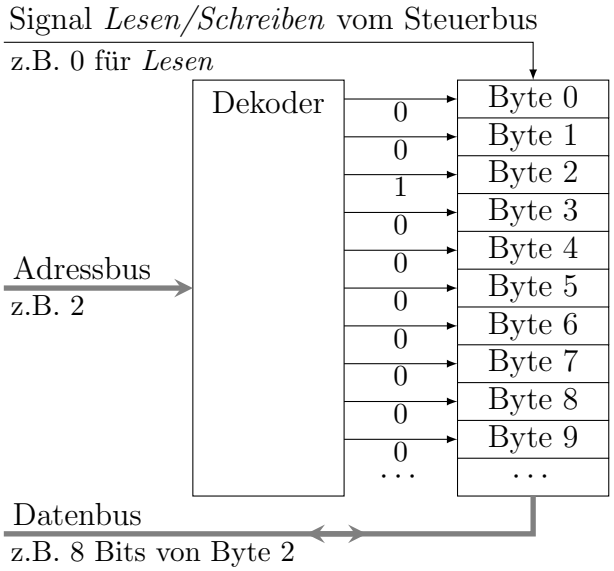


Abbildung 1.4: Adressierung von Speicherzellen.

Es ist wichtig, das Konzept von Adresse und Speicherinhalt gut zu verstehen. Dieses Wissen wird später für die C-Programmierung mit Zeigern (engl. pointer) benötigt.

1.3 Duale, hexadezimale und andere Zahlensysteme

In diesem Abschnitt wird eine Einführung in verschiedene Zahlensysteme gegeben. Der Fokus liegt dabei in den für die Computertechnik nützlichen Zahlensystemen. Das sind das *duale* (binäre), *oktale* und *hexadezimale* (sedezimale) Zahlensystem. Nach einem kurzen Überblick soll das Zählen im Dezimalsystem als Grundlage für die Herleitung anderer Zahlensysteme dienen.

1.3.1 Überblick

Zunächst betrachten wir das wohl bekannteste und verbreitetste Zahlensystem, das Dezimalsystem. Das Dezimalsystem hat die Basis 10, das heißt, für jede Ziffer stehen zehn unterschiedliche Symbole zur Verfügung (0, 1, 2, 3, 4, 5, 6, 7, 8 und 9).

Grundsätzlich kommt aber jede ganze Zahl größer 1 als Basis für ein Zahlensystem in Frage. Nun muss man sich natürlich die Frage nach dem Sinn und Zweck einer gewählten Basis stellen. Der Grund, dass sich das Dezimalsystem so verbreitet hat, liegt vermutlich nicht in tiefen mathematischen Überlegungen, sondern einfach in der Tatsache, dass wir zehn Finger haben.

Das grundlegende Zahlensystem in der Computertechnik ist das Dualsystem (auch Binärsystem genannt). Eine elementare Speicherzelle eines Computers kann nur zwei Zustände annehmen: Spannung oder Nichtspannung, eins oder null, high oder low. (Es gibt wenige Ausnahmen, z.B. Analogrechner.) Daher bietet es sich an, durch eine Verkettung von elementaren Speicherzellen eine duale Zahl zu erzeugen.

Duale Zahlen werden schnell sehr lang. Für die dezimale 9 werden dual bereits vier Ziffern benötigt, für die 99 sieben Ziffern etc. Um eine bessere Übersicht zu erlangen, wurden in der Praxis drei Dualziffern zu einer Oktalziffer, bzw. vier Dualziffern zu einer hexadezimalen Ziffer zusammengefasst. Wie das im Einzelnen geschieht, wird in den folgenden Abschnitten erläutert.

1.3.2 Zählen im Dezimalsystem

Die (betragsmäßig) kleinste Zahl ist die Zahl Null (Symbol 0). Fängt man nun an zu zählen, so benutzt man der Reihe nach die Symbole 1, 2, 3 u.s.w. bis man schließlich bei der 9 angelangt. Will man über die 9 hinaus zählen, bedient man sich einer weiteren Ziffer, die der anderen vorangestellt wird. Die vordere erhält den Wert 1, während die hintere auf 0 gesetzt wird (9→10). Damit erhält die vordere Ziffer die Gewichtung 10, die hintere behält ihre einfache Gewichtung.

Mit diesen beiden Ziffern schafft man es nun bis 99 zu zählen. Will man weiterzählen, so schafft man sich eine weitere Ziffer mit der Gewichtung 100. Die neue Ziffer wird den anderen vorangestellt und erhält den Wert 1, die anderen beiden werden auf 0 gesetzt (99→100).

Aus diesen Überlegungen leitet sich folgende Erkenntnis ab: Die zuerst verwendete Ziffer (ganz rechts) hat die Gewichtung 1. Alle Ziffern davor haben jeweils die zehnfache Gewichtung ihres hinteren (rechten) Nachbarn.

Sollen Bruchteile einer Zahl dargestellt werden, so verwendet man eine oder mehrere Ziffern, die durch ein Komma getrennt hinter die anderen Ziffern (rechts)

angehängt werden. Hier gilt der umgekehrte Zusammenhang, jede Ziffer nach dem Komma hat jeweils ein Zehntel der Gewichtung seines vorderen (linken) Nachbarn.

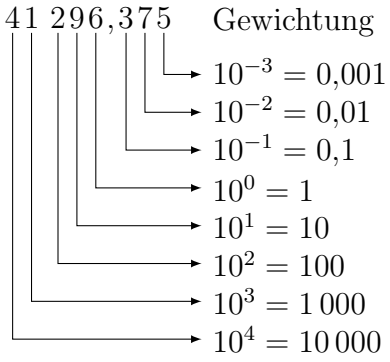


Abbildung 1.5: Gewichtung der Ziffern im Dezimalsystem.

Ein wichtiges Merkmal ist die Tatsache, dass sich die Gewichtung zweier benachbarter Ziffern immer um den Faktor zehn unterscheidet, was seine Ursache im Zeichenumfang des Zahlensystems (hier 0 bis 9) und damit in der Basis des Zahlensystems hat.

1.3.3 Zählen in anderen Zahlensystemen

Das uns bekannte Dezimalsystem hat die Basis zehn. Das heißt, es stehen zehn Symbole für die Ziffern zur Verfügung, und die Gewichtung zweier benachbarter Ziffern unterschiedet sich ebenfalls um den Faktor zehn. Im Vergleich zum Dezimalsystem haben andere Zahlensysteme einen anderen Symbolumfang für die Ziffern und damit einhergehend eine andere Gewichtung der einzelnen Ziffern.

Nehmen wir z.B. das *oktale* Zahlensystem (Basis: 8), so werden statt der gewohnten zehn Symbole nur acht verwendet (0, 1, 2, 3, 4, 5, 6 und 7). Wird nun beim Zählen die Zahl 7 überschritten, so wird eine zweite Ziffer mit der Gewichtung acht der anderen Ziffer vorangestellt. Die vordere Ziffer erhält den Wert eins, während die hintere auf null gesetzt wird (7→10). Die so entstandene Zahl sieht wie eine zehn im Dezimalsystem aus, beschreibt aber nur einen Wert von acht.

Die höchste im Oktalsystem mit zwei Ziffern darstellbare Zahl ist die 77 was einem dezimalen Wert von 63 entspricht. Wird dieser Wert um eins erhöht, so werden beide Ziffern auf null gesetzt und eine eins als dritte Ziffer den anderen vorangestellt (77→100). Die neue Ziffer hat demnach eine Gewichtung von 64 (dezimal ausgedrückt).

In Tabelle 1.1 sind die Zahlen von null bis zwanzig in verschiedenen Zahlensystemen zusammengetragen. Auf das duale, oktale und hexadezimale Zahlensystem wird im Folgenden eingegangen. Das terziale und septale Zahlensystem ist nur als ein mathematisches Beispiel aufgeführt, beide haben in der Praxis keine Bedeutung (können getrost vergessen werden). Das hexadezimale Zahlensystem umfasst 16 Symbole, von denen die letzten 6 durch die Buchstaben A bis F dargestellt werden.

dezimal	dual/binär	oktal	hexadezimal	terzial	septal
Basis 10	Basis 2	Basis 8	Basis 16	Basis 3	Basis 7
0	0	0	0	0	0
1	1	1	1	1	1
2	10	2	2	2	2
3	11	3	3	10	3
4	100	4	4	11	4
5	101	5	5	12	5
6	110	6	6	20	6
7	111	7	7	21	10
8	1000	10	8	22	11
9	1001	11	9	100	12
10	1010	12	A	101	13
11	1011	13	B	102	14
12	1100	14	C	110	15
13	1101	15	D	111	16
14	1110	16	E	112	20
15	1111	17	F	120	21
16	10000	20	10	121	22
17	10001	21	11	122	23
18	10010	22	12	200	24
19	10011	23	13	201	25
20	10100	24	14	202	26

Tabelle 1.1: Die Zahlen von null bis zwanzig in verschiedenen Zahlensystemen.

1.3.4 Geeignete Zahlensysteme in der Computertechnik

Wie bereits erwähnt, können die elementaren Speicherzellen fast aller Computer nur zwei Zustände annehmen (...dafür gibt es aber reichlich davon). Diese Begrenzung auf zwei Zustände führten zu der Bevorzugung des dualen Zahlensystems in der Computertechnik. Acht elementare Speicherzellen (8 Bit = 1 Byte) ergeben eine duale Zahl mit dem Wertebereich von 0 bis 255. Mit sechzehn Bit (2 Byte) erreicht man bereits die Zahl 65 535, mit 32 Bit (4 Byte) die Zahl 4 294 967 295, mit 64 Bit (8 Byte) die Zahl 18 446 744 073 709 551 615 etc. (Rechnet man mit vorzeichenbehafteten Zahlen, so wird der Zahlenberiech um null herum gelegt.)

Nun ist es bei der Darstellung von dualen Zahlen unangenehm, dass gegenüber dezimalen Zahlen drei bis viermal soviel Ziffern benötigt werden. Möchte man größere Speicherblöcke betrachten, sieht man sich schnell einem Wust von Einsen und Nullen gegenübergestellt. Um dem entgegenzuwirken, kam die Idee, einfach jeweils drei oder vier Bits zu einer Ziffer zusammenzufassen. Mit drei Bits können Zahlen von null bis sieben wiedergegeben werden, was dem Zeichnumfang des *oktalen* Zahlensystems entspricht. In vier Bits können Zahlen von null bis fünfzehn gespeichert werden, was dem Symbolumfang des *hexadezimalen* Zahlensystems entspricht.

1.3.5 Umrechnung zwischen den Zahlensystemen

Anmerkung: Um das Zahlensystem einer Zahl anzudeuten wird die Basis als Suffix (tiefgestellt) angehängt. Beispiele: 23_{10}, 100101_2, $6F_{16}$

Umrechnung von einem beliebigen ins dezimale Zahlensystem

Die zu konvertierende Zahl sei wie folgt dargestellt: $\cdots z_3 z_2 z_1 z_0, z_{-1} z_{-2} z_{-3} \cdots$ Mit B als der Basis der zu konvertierenden Zahl ergibt sich für a als die umgerechnete Zahl:

$$a = \sum_i z_i B^i$$

Mit anderen Worten: Um eine Zahl eines beliebigen Zahlensystems dezimal darzustellen, addiere man das Produkt jeder Ziffer mit der jeweiligen Gewichtung der Ziffer. Beispiele:

- $6\mathrm{F}_{16} = 6{\cdot}16 + 15{\cdot}1 = 111_{10}$
- $10011_2 = 1{\cdot}16 + 0{\cdot}8 + 0{\cdot}4 + 1{\cdot}2 + 1{\cdot}1 = 19_{10}$
- $1100{,}1101_2 = 8 + 4 + 0{,}5 + 0{,}25 + 0{,}0625 = 12{,}8125_{10}$

Umrechnung einer Dezimalzahl in ein beliebiges Zahlensystem

Diese Umrechnung muss getrennt für den ganzzahligen Teil und den Nachkommateil betrachtet werden.

Umrechnung des ganzzahligen Teils. Man führe eine ganzzahlige Division der Dezimalzahl durch die Basis des Zielsystems durch. Der Rest der Division Stellt den Wert der kleinsten ganzzahligen Ziffer im neuen Zahlensystem dar. Der Quotient dient als Dividend zur Berechnung der nächsten Ziffer nach dem gleichen Schema. Der Rest der zweiten Division wird links vor die zuvor berechnete Ziffer geschrieben. Es werden weitere Divisionen durchgeführt, bis der Quotient null ergibt.

Beispiel: $1234_{10} \rightarrow$ hexadezimal

1234/16 =	77	Rest 2	↑
77/16 =	4	Rest 13 (D)	$4\mathrm{D}2_{16}$
4/16 =	0	Rest 4	

Beispiel: $25_{10} \rightarrow$ dual

25/2 =	12	Rest 1	↑
12/2 =	6	Rest 0	
6/2 =	3	Rest 0	11001_2
3/2 =	1	Rest 1	
1/2 =	0	Rest 1	

Umrechnung des Nachkommateils. Man multipliziere den Nachkommateil mit der Basis des Zielsystems. Der ganzzahlige Teil ergibt die erste Ziffer nach dem Komma. Der Nachkommateil des zuvor berechneten Produktes dient als Ausgangspunkt für die Berechnung der nächsten Ziffer. Dies wird fortgeführt, bis die Umrechnung abgeschlossen ist. Da der Nachkommateil häufig auch nach vielen berechneten Ziffern nicht auf Null geht, muss in solchen Fällen die Umrechnung bei Erreichen der gewünschten Genauigkeit abgebrochen werden.

Beispiel: $0{,}375_{10} \rightarrow$ dual

$$\begin{array}{rll} 0{,}375 \cdot 2 = 0{,}75 & \text{ganzzahliger Teil: } 0 & \Big| \\ 0{,}75 \cdot 2 = 1{,}5 & \text{ganzzahliger Teil: } 1 & \Big| \\ 0{,}5 \cdot 2 = 1 & \text{ganzzahliger Teil: } 1 & \downarrow = 0{,}011_2 \end{array}$$

Beispiel: $0{,}1_{10} \rightarrow$ oktal

$$\begin{array}{rll} 0{,}1 \cdot 8 = 0{,}8 & \text{ganzzahliger Teil: } 0 & \Big| \\ 0{,}8 \cdot 8 = 6{,}4 & \text{ganzzahliger Teil: } 6 & \Big| \\ 0{,}4 \cdot 8 = 3{,}2 & \text{ganzzahliger Teil: } 3 & \Big| \\ 0{,}2 \cdot 8 = 1{,}6 & \text{ganzzahliger Teil: } 1 & \Big| \\ 0{,}6 \cdot 8 = 4{,}8 & \text{ganzzahliger Teil: } 4 & \Big| \\ 0{,}8 \cdot 8 = 6{,}4 & \text{ganzzahliger Teil: } 6 & \Big| \\ 0{,}4 \cdot 8 = 3{,}2 & \text{ganzzahliger Teil: } 3 & \downarrow = 0{,}0\overline{6314}_8 \end{array}$$

Umrechnung dualer Zahlen in oktale und hexadezimale Zahlen

Drei Ziffern im dualen Zahlensystems lassen sich zu einer oktalen Ziffer zusammenfassen während vier Ziffern des dualen Zahlensystems eine hexadezimale Ziffer ergeben. Will man nun eine duale Zahl in eine oktale umwandeln, so fasst man ausgehend vom Dezimalkomma die Dualziffern in Blöcken von drei zusammen. Jeder Dreierblock wird dann gemäß Tabelle 1.2 umgewandelt. Soll eine Dualzahl in eine hexadezimale Zahl umgewandelt werden, so werden wieder vom Dezimalkomma aus die Dualziffern diesmal in Blöcken zu vier gruppiert. Die Viererblöcke werden dann gemäß der Tabelle in hexadezimale Ziffern umgewandelt.

dual	oktal
000	0
001	1
010	2
011	3
100	4
101	5
110	6
111	7

dual	hexadezimal
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7

dual	hexadezimal
1000	8
1001	9
1010	A
1011	B
1100	C
1101	D
1110	E
1111	F

Tabelle 1.2: Umrechnung zwischen dualen und oktalen, bzw. dualen und hexadezimalen Zahlen.

Beispiele:

- $1010100100_2 = 001\ 010\ 100\ 100 = 1244_8$
- $10010101100_2 = 0100\ 1010\ 1100 = 4\mathrm{AC}_{16}$
- $10011101{,}00101_2 = 1001\ 1101\ ,\ 0010\ 1000 = 9\mathrm{D}{,}28_{16}$

Umrechnung oktaler und hexadezimaler Zahlen in Dualzahlen

Bei der Umrechnung von oktalen in duale Zahlen brauchen einfach nur alle oktalen Ziffern gemäß Tabelle 1.2 in Dreierblöcke von dualen Ziffern umgewandelt werden.

Das gleiche gilt für die Umwandlung hexadezimaler Zahlen, dann aber in Blöcke von vier dualen Ziffern.

Beispiel:

- $567{,}4_8$ = 101 110 111 , 100 = $101110111{,}1_2$
- $1D4{,}6_{16}$ = 0001 1101 0100 , 0110 = $111010100{,}011_2$

1.3.6 Negative Zahlen

Auf dem Papier werden negative Zahlen durch ein Minuszeichen vor der Zahl angedeutet. Das positive Vorzeichen wird normalerweise nicht dargestellt, aber als implizit angenommen. Bei dieser Schreibweise gibt es für die Null eine Redundanz: Die Null mit positivem Vorzeichen ist identisch zur Null mit negativem Vorzeichen. Nicht zuletzt deshalb wird in der Computertechnik eine spezielle Form für negative Zahlen verwendet:

Das Zweierkomplement

Beim Zweierkomplement dient das erste Bit als Vorzeichen. Eine null steht für ein positives Vorzeichen, eine eins steht für ein negatives Vorzeichen. Die positiven Zahlen werden wie gewohnt im dualen Zahlensystem wiedergegeben. Für die negativen Zahlen wird das Zweierkomplement gebildet: Erst werden alle Einsen in Nullen und alle Nullen in Einsen umgewandelt, danach wird die Zahl um eins erhöht.

Beispiel: 8-Bit Zahlen im Zweierkomplement

- -4_{10} = -0000 0100_2 → 1111 1011 → 1111 1100
- -75_{10} = -0100 1011_2 → 1011 0100 → 1011 0101

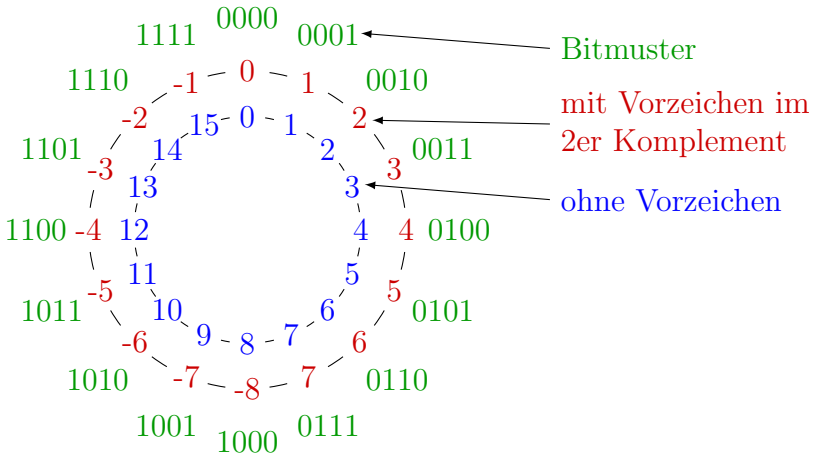


Abbildung 1.6: Darstellung von positiven und negativen Zahlen für vier Bit.

Umgekehrt gilt das gleiche: Um den Wert einer negativen Zweierkomplementzahl zu entschlüsseln, müssen erst alle Ziffern umgekehrt werden, bevor dann die entstandene Zahl um eins erhöht wird.

Beispiel: 8-Bit Zahlen im Zweierkomplement

- 1111 1100 → 0000 0011 → -0000 0100_2 = -4_{10}
- 1011 0101 → 0100 1010 → -0100 1011_2 = -75_{10}

1.4 Programmiersprachen

1.4.1 Die Sprache der CPU: Maschinensprache

Der Inhalt von Speicherzellen kann grob in zwei Kategorien gegliedert werden: Befehle und Daten. Während die Daten einfach nur abgelegt und später wieder abgerufen werden, steuern Befehle die Aktionen der CPU. Als Maschinensprache bezeichnet man die Befehle in Form von dualen Zahlen, welche die CPU steuern.

Alte CPUs lesen der Reihe nach die Befehle aus dem Speicher und führen sie aus, einen nach dem anderen. Moderne Rechner lesen während der Ausführung eines Befehls schon den nächsten, bzw. mehrere folgende Befehle ein, und beginnen parallel auch diese auszuführen. Das kann natürlich beliebig kompliziert werden, zumal die Befehle häufig voneinander abhängen. Die Zukunft von Computern liegt neben der weiteren Beschleunigen vor allem in der Verbesserung der parallelen Verarbeitung von Prozessen.

Der Inhalt eines Befehls für eine CPU ist sehr primitiv. Soll z.B. der Name 'Robert Heß' auf dem Bildschirm dargestellt werden, darf nicht erwartet werden, das dies mit einem einzigen Befehl zu erreichen sei. Selbst ein Befehl pro Buchstabe ist bei weitem zu wenig. Typische Befehle an die CPU sind: Speichere die Zahl x in deinen Zwischenspeicher (Register), oder: Ermittle, ob die Zahl x größer als die Zahl y ist. Weiter gibt es Sprungbefehle, die der CPU mitteilen, dass die Abarbeitung von Befehlen an einer anderen Stelle fortgeführt werden soll.

Die Kunst liegt nun in der geschickten Verschachtelung dieser Befehle. Die Befehle einer CPU sind recht primitiv, dafür werden sie aber extrem schnell bearbeitet. So kann z.B. die Berechnung des Tangens eines Winkels in weniger als einer Mikrosekunde auf zehn und mehr Stellen genähert werden.

1.4.2 Höhere Programmiersprachen

Wären alle Programmierer dazu verpflichtet, ihre Programme in Maschinensprache zu schreiben, so wären wir heute nur von recht primitiver Software umgeben. Der Aufwand wäre einfach zu groß.

Um diese Hürde zu überwinden wurden in der Geschichte der Computer recht schnell sogenannte *höhere Programmiersprachen* entwickelt. Ein Programm, das in einer höheren Programmiersprache geschrieben wurde, kann nicht direkt von der CPU gelesen werden. Es wird eine geeignete Software dazwischen geschaltet, welche die Befehle der höheren Sprache in ganze Befehlspakete in Maschinensprache überträgt, die dann von der CPU gelesen und ausgeführt werden können.

Über die Jahre haben sich viele Programmiersprachen entwickelt. Zum Teil haben sie unterschiedliche Aufgabenschwerpunkte (z.B. COBOL in der Geschäftswelt und Fortran in Wissenschaft und Technik), oder sie sind Herstellerspezifisch (z.B. Makrosprachen innerhalb von Officepaketen). Andere Sprachen entstanden um Zuge neuer Computertechniken (z.B. Java als rein objektorientierte Sprache).

1.4.3 Übersetzer und Interpreter

Programmiersprachen lassen sich in zwei Gruppen einteilen: *Interpreter-* und *Übersetzer-* (*Compiler-*) Sprachen.

Ein in einer Interpreter-Sprache geschriebenes Programm wird bei der Ausführung zeilenweise gelesen und in Maschinensprache übertragen. Ein solches Programm kann unmittelbar gestartet werden. Fehler werden erst bei Erreichen der entsprechenden Zeile entdeckt. Die alten Dialekte von Basic waren nach diesem Schema aufgebaut. Eine solche Sprache braucht zur Ausführung immer den entsprechenden Interpreter.

Ein in einer Übersetzersprache geschriebenes Programm wird vor Beginn der Ausführung komplett in maschinenlesbare Befehle übersetzt. Alle Eingabefehler müssen vor dem ersten Ausführen behoben werden. Der Übersetzungsvorgang kann einige Zeit dauern (für kleine Programme Sekunden, für große professionelle Softwarepakete Stunden oder Tage). Der große Vorteil dieser Sprachen liegt in der Tatsache, dass während der Ausführung keine Zeit für das Übersetzen der Befehle benötigt wird. Außerdem wird der Übersetzer (Compiler) für die eigentliche Ausführung des Programms nicht mehr benötigt, d.h. die geschriebene Software wird ohne Übersetzer ausgeliefert.

	Übersetzer	**Interpreter**
Übertragung der Befehle	vor Programmstart	während der Ausführung
Erster Start der Ausführung	erst nach der Übersetzung	sofort
Ausführungs-geschwindigkeit	schneller	langsamer
Erkennung der Eingabefehler	zu Beginn	erst während der Ausführung
Ausführung	ohne Übersetzer	nur mit Interpreter

Tabelle 1.3: Vergleich von Übersetzer und Interpreter.

Im professionellen Bereich werden fast ausschließlich Übersetzersprachen verwendet. Eine weit verbreitete ist die Programmiersprache C, die wir in dieser Vorlesung kennenlernen.

1.5 Aufgaben

Aufgabe 1.1: Wandeln Sie folgende Zahlen in das duale/binäre Zahlensystem:

a) 23_{10} b) 45_{10} c) 129_{10}
d) $0{,}5_{10}$ e) $3{,}75_{10}$ f) $17{,}0625_{10}$
g) $99{,}9_{10}$ h) $0{,}1_{10}$ i) $0{,}01_{10}$

Aufgabe 1.2: Wandeln Sie in das dezimale Zahlensystem um:

a) $1F_{16}$ b) $1001\ 1001_2$ c) 567_8
d) $0{,}1_2$ e) $0{,}1_8$ f) $0{,}1_{16}$

Aufgabe 1.3: Stelle Sie im 8-Bit Zweierkomplement dar:
a) -101_2 b) -1_{10} c) -64_{10}

Aufgabe 1.4: Für Fortgeschrittene (nicht Klausurrelevant):
a) $23_{10} \rightarrow$ terzial b) $49_{10} \rightarrow$ septal
c) $0{,}5_{10} \rightarrow$ terzial d) $0,\overline{142857}_{10} \rightarrow$ septal

Aufgabe 1.5: Erläutern Sie die Begriffe *Maschinensprache* und *höhere Programmiersprache*.

Aufgabe 1.6: Zeigen Sie die Vor- und Nachteile von *Übersetzer*-Sprachen gegenüber *Interpretierer*-Sprachen in Bezug auf *Zeitpunkt* der Übersetzung, *Ausführungsgeschwindigkeit* und *benötigte Programme* zum Zeitpunkt der Ausführung.

Kapitel 2

Die ersten C-Programme

Der erste Einstieg in die C-Programmierung soll anhand einiger Beispielprogramme erfolgen. Das erste Programm ist der Klassiker: „Hello World“. Danach beschäftigen wir uns der Reihe nach mit Variablen, Wiederholungen und Gleitkommazahlen.

Das die Programmierumgebungen recht unterschiedlich ausfallen, wird hier nicht darauf eingegangen. Siehe hierfür die Beschreibung für den verwendeten Compiler.

2.1 Beispiel: „Hello World“

Dies ist das erste Programm in C. Es enthält alle nötigen Elemente, um den Text *hello world* auf den Bildschirm auszugeben.

2.1.1 Der Quellcode

```
#include <stdio.h>

int main()
{
    printf("hello world\n");

    return 0;
}
```

2.1.2 Ausgabe auf dem Bildschirm

Wenn Sie dieses Programm starten erscheint folgende Ausgabe auf dem Bildschirm:

```
hello world
```

2.1.3 Erläuterungen

Spracherweiterungen mit *#include*. Der Kern der Sprache C ist äußerst begrenzt. Selbst die einfachste Ein- und Ausgabe ist nicht möglich. Aber C bietet die Möglichkeit beliebig erweitert zu werden. So gehören zu C selbstverständlich Erweiterungen zur Ausgabe auf dem Bildschirm und zur Abfrage der Tastatur. Mit der Zeile **#include**... werden dem Programm solche Erweiterungen

mitgeteilt. Alle mit Doppelkreuz eingeleiteten Befehle werden vor der eigentlichen Übersetzung des Quellcodes vom Preprozessor („Vor-Verarbeiter“) bearbeitet. Bei dem Include-Befehl ersetzt der Preprozessor die Zeile mit dem Inhalt der angegebenen Datei (hier stdio.h). Siehe Abschnitt 7.3 auf Seite 72 für mehr Details.

Befehle zur Ein- und Ausgabe: *stdio.h*. Die Bezeichnung *stdio* steht für *standard input output* und beinhaltet standardisierte Funktionen für die Ein- und Ausgabe. In der Datei *stdio.h* wird nur das Aussehen der neuen Sprachelemente beschrieben (z.B. *printf*). Die eigentlichen Befehle werden erst beim Verknüpfen der Progammstücke durch den *Linker* hinzugefügt, siehe Abschnitt 7.1 auf Seite 70.

Das Hauptprogramm *main*. Ein normales C-Programm enthält das Schlüsselwort *main* als die Stelle, an der das Programm gestartet wird. (Für Fensteranwendungen wird es durch das Schlüsselwort *WinMain* ersetzt.) In den Klammern hinter dem Wort *main* steht, welche Parameter dem Programm übergeben werden. In dem Beispiel werden keine Werte übergeben. Vor dem Wort *main* wird angegeben, was das Programm zurück gibt. In unserem Fall soll eine ganze Zahl vom Typ *int* zurückgegeben werden. Auf die Rückgabewerte wird im Detail im Abschnitt 5.1.3 auf Seite 57 eingegangen.

Zusammenfassung von Befehlen durch geschweifte Klammern {}. Die geschweiften Klammern umfassen alle Befehle des Hauptprogramms *main*. Befehle außerhalb dieser Klammern sind unzulässig.

Formartierte Ausgabe mit *printf*. Die Anweisung *printf* steht für *print formatted*, zu deutsch *schreibe formatiert*. Die *printf*-Anweisung wird hier in einer einfachen Form verwendet. Sie schreibt die Zeichenkette, die in Klammern übergeben wird, auf den Bildschirm. Für mehr Details siehe Abschnitt A.1.1 auf Seite 81.

Zeichenketten (Strings). Eine Zeichenkette kann ein Wort, ein Satz oder jede beliebige *Kette* von Zeichen sein. Eine Zeichenkette wird durch Doppelanführungszeichen („Gänsefüßchen“) begrenzt.

In unserem Beispiel wird ein Sonderzeichen verwendet. Das '\n' steht für einen Zeilenumbruch. Auch Wenn das Sonderzeichen '\n' in der Zeichenkette durch zwei Zeichen wiedergegeben wird, wird es nur als ein einzelnes Zeichen interpretiert, und es wird im ausführbaren Programm nur eine Speicherstelle (ein Byte) benötigt.

Das Semikolon ; am Ende einer Anweisung. In C wird das Ende einer Anweisung durch ein Semikolon angezeigt. Fehlt das Semikolon, so gibt es von dem Übersertzer (dem *Compiler*) eine Fehlermeldung.

Beenden einer Routine mit *return*. Wie bereits erwähnt, wurde im Kopf des Hauptprogramms festgelegt, dass eine ganze Zahl vom Typ *int* zurückgegeben werden soll. Mit der Anweisung *return* wird das Programm beendet, und der Wert dahinter zurückgegeben. In diesem Fall wird der Wert null zurückgegeben.

2.2 Beispiel: Variablen

In diesem Beispielprogramm lernen Sie ein wichtiges Element einer jeden Programmiersprache kennen: Die Verwendung von *Variablen*. Es wird eine Variable definiert, es wird ihr ein Wert zugewiesen, danach wird der Wert verändert, und schließlich wird das Ergebnis auf dem Bildschirm ausgegeben.

2.2.1 Der Quellcode

```
#include <stdio.h>

/* Verwendung von Variablen */
int main()
{
    int a;          /* Definition einer Variable */

    a = 7;          /* Zuweisung eines Werts */
    a = a+4;        /* Verwendung eines Operators */

    printf("Ergebnis: %d\n", a);

    return 0;
}
```

2.2.2 Ausgabe auf dem Bildschirm

Es erscheint folgende Ausgabe auf dem Bildschirm:

```
Ergebnis: 11
```

2.2.3 Erläuterungen

Kommentare. Kommentare werden mit Schrägstrich und Stern eingeleitet (/*), können sich über mehrere Zeilen erstrecken und werden mit Stern und Schrägstrich beendet (*/). Kommentare dienen der übersichtlichen Gestaltung des Quellcodes: Andere Programmierer können sich leichter in Ihre Programme eindenken, und auch Sie selbst erinnern sich nach längerer Zeit besser an Ihre früheren Werke.

Definition von Variablen. In C müssen Variablen vor der Verwendung definiert werden. Zuerst wird der Datentyp festgelegt. Hier verwenden wir *int*, das ist eine ganze, vorzeichenbehaftete Zahl. Dahinter, auf der rechten Seite, steht ein frei gewählter Variablenname, der die definierte Variable bezeichnet. Die Definition wird mit einem Semikolon abgeschlossen. Nach dieser Zeile ist dem Programm die Variable *a* vom Typ *int* bekannt. Eine ausführliche Beschreibung zum Thema Variablen findet sich im Abschnitt 3.2 auf Seite 29.

Einer Variablen einen Wert zuweisen. Die Zuweisung eines Wertes an eine Variable erfolgt mit dem Gleichheitszeichen. Links vom Gleichheitszeichen steht die Variable, der ein Wert zugewiesen werden soll. Rechts davon steht ein Ausdruck, dessen Ergebnis der Variable zugewiesen wird. Die Zuweisung wird wieder mit einem Semikolon abgeschlossen.

Operatoren zur Verarbeitung von Daten. Operatoren dienen der Manipulation von Daten. Der Kern von C kennt neben vielen anderen Operatoren die vier Grundrechenarten (+, -, *, /) und die Restdivision (%). Des Weiteren können beliebig runde Klammern gesetzt werden. Es gilt wie gewohnt Punkt- vor Strichrechnung. In diesem Beispiel wird zu dem Inhalt von *a* (der Wert 7) die Zahl 4 dazuaddiert. Das Ergebnis 11 wird dann mit dem Gleichheitszeichen wieder der Variable *a* zugewiesen. (Hier wird deutlich, dass in C ein Gleichheitszeichen nicht im mathematischen Sinne der Gleichheit verstanden werden darf. Es handelt sich um eine Zuweisung.) Für mehr Details siehe Abschnitt 3.4 auf Seite 34.

Anzeige des Inhalts von Variablen. Der *printf*-Befehl wurde bereits im Beispiel *HelloWorld* eingeführt, siehe Abschnitt 2.1. Hier wird der Befehl verwendet, um den Inhalt der Variable *a* anzuzeigen. Der *printf*-Befehl erwartet als ersten Parameter eine Zeichenkette, der die Formatierung der Ausgabe festlegt. Für die Ausgabe von Variablen des Typs *int* wird der Platzhalter *%d* in die Zeichenkette eingefügt. Wenn der *printf*-Befehl einen solchen Platzhalter vorfindet, erwartet er nach der Zeichenkette durch ein Komma getrennt den Namen der Variablen. In unserem Fall wird entsprechend die Variable *a* hinter der Zeichenkette eingefügt. Eine ausführliche Beschreibung des *printf*-Befehls findet sich im Anhang A.1.1 auf Seite 81.

2.3 Beispiel: Wiederholungen

In diesem Beispielprogramm lernen Sie ein weiteres wichtiges Element von Programmiersprachen kennen: Schleifen. Sie dienen dazu Programmstücke mehrfach auszuführen. Des Weiteren wird das Thema Variablen vertieft. Das folgende Programm erstellt eine Tabelle, mit der Temperaturen von Grad Fahrenheit (°F) in Grad Celsius (°C) umgerechnet werden.

2.3.1 Der Quellcode

```
#include <stdio.h>

/* Umwandlung von Fahrenheit in Celsius */
int main()
{
    int fahr, celsius;
    int lower, upper, step;

    lower = 0;          /* untere Grenze der Temperatur */
    upper = 300;        /* obere Grenze */
    step = 20;          /* Schrittweite */

    /* Ausgabe einer Überschrift */
    printf("Fahr.\tCelsius\n");

    /* Erstellung der Tabelle */
    fahr = lower;
    while(fahr<=upper) {
        celsius = 5*(fahr-32)/9;
```

```
        printf("%d\t%d\n", fahr, celsius);
        fahr = fahr+step;
    }
    return 0;
}
```

2.3.2 Ausgabe auf dem Bildschirm

Das Programm erzeugt folgende Ausgabe auf dem Bildschirm:

```
Fahr.   Celsius
0       -17
20      -6
40      4
60      15
80      26
100     37
120     48
140     60
160     71
180     82
200     93
220     104
240     115
260     126
280     137
300     148
```

2.3.3 Erläuterungen

Definition von mehreren Variablen. Im Beispiel *Variablen* (Abschnitt 2.2) haben wir bereits die Definition einer einzelnen Variable kennengelernt. Sollen mehrere Variablen definiert werden, so stehen zwei Möglichkeiten zur Verfügung: a) Es wird für jede Variable eine extra Zeile mit Datentyp und Variablennamen eingefügt. b) Bei mehreren Variablen gleichen Typs wird der Datentyp einmal angegeben gefolgt von einer durch Kommata getrennten Liste von Variablennamen. In beiden Fällen wird die Definition durch ein Semikolon abgeschlossen. In dem hier aufgeführten Beispiel wurde eine Kombination dieser Möglichkeiten verwendet: Es werden fünf Variablen auf zwei Zeilen verteilt definiert.

Sonderzeichen. Beim Drucken von Zeichenketten werden einige Zeichenfolgen als *Sonderzeichen* interpretiert. Bereits beschrieben wurde das Sonderzeichen '\n', welches einen Zeilenvorschub bewirkt. In diesem Beispiel wird die Zeichenfolge '\t' verwendet, welche als ein Tabulatorzeichen bewertet wird. Der Tabulator dient der Ausrichtung von Spalten. Es gibt noch eine Reihe weiterer Sonderzeichen, die in Tabelle 3.2 auf Seite 33 zusammengefasst sind.

Wiederholen von Programmstücken mit *while*. In diesem Beispielprogramm wird eine Tabelle mit 16 Zeilen erstellt. Es wäre möglich, die 16 Zeilen durch

16 *printf*-Befehle auszugeben. Da aber die Zeilen in der Tabelle alle recht ähnlich aussehen, bietet es sich an, ein Programmstück mehrfach mit jeweils geänderten Bedingungen auszuführen. Für diesen Zweck bieten sich Schleifen an.

Der Befehl *while* sorgt dafür, dass die Befehle, die in geschweiften Klammern dahinter aufgeführt sind, mehrfach ausgeführt werden. In der runden Klammer hinter dem Wort *while* steht die Schleifenbedingung. Die Schleife wird solange ausgeführt, wie diese Schleifenbedingung erfüllt ist und den Wert *wahr* ergibt. In diesem Beispielprogramm soll die Schleife ausgeführt werden, solange die Variable *fahr* kleiner oder gleich der Variable *upper* ist. Das heißt hier, erst wenn die Variable *fahr* einen Wert größer 300 annimmt, ergibt die Schleifenbedingung den Wert *nicht wahr* und die Schleife wird beendet.

Bei der Programmierung von Schleifen ist darauf zu achten, dass die Schleife auch zu einem Ende kommt. Innerhalb der Schleife muss Einfluss auf das Ergebnis der Schleifenbedingung genommen werden. In diesem Beispielprogramm wird am Ende des Schleifeninhalts die Variable *fahr* um den Wert der Variable *step*, hier 20, erhöht. Dadurch hat die Schleife die Chance zu einem Ende zu kommen. Für mehr Details zum Thema Schleifen siehe Abschnitt 4.3 auf Seite 48.

Vergleichsoperatoren. Ein Vergleichsoperator vergleicht zwei Operanden und liefert als Ergebnis den Wahrheitswert *wahr* oder *nicht wahr* (*falsch*) zurück. in unserem Beispiel wird geprüft, ob die Variable *fahr* kleiner oder gleich der Variable *upper* ist. Solange dies der Falls ist ergibt der Ausdruck *wahr*, und die Schleife wird weiter ausgeführt. Für mehr Details zu Vergleichsoperatoren siehe Abschnitt 3.4.2 auf Seite 35.

Verknüpfte Operatoren. In C können Operatoren beliebig verknüpft werden. Es gilt Punkt- vor Strichrechnung. Soll, wie in diesem Beispiel, eine Strichrechnung bevorzugt werden, so muss eine Klammer gesetzt werden. Da dieses Beispielprogramm nur ganze Zahlen verarbeitet, werden alle Ergebnisse, auch die Zwischenergebnisse, auf ganze Zahlen abgerundet. (Genauer gesagt wird immer in Richtung null gerundet, das heißt, positive Zahlen werden abgerundet, negative Zahlen werden aufgerundet.)

Anzeige des Inhalts von Variablen. Im Beispiel *Variablen* (Abschnitt 2.2) wurde bereits gezeigt, wie der Inhalt einer einzelnen Variable angezeigt wird. Um den Inhalt von zwei Variablen anzuzeigen, werden in der Zeichenkette zwei Platzhalter für den entsprechenden Typ (hier %d) eingefügt. Hinter der Zeichenkette müssen dann die Variablen in der Reihenfolge durch Kommata getrennt angefügt werden wie sie in der Zeichenkette durch die Platzhalter angekündigt wurden. Der *printf*-Befehl arbeitet die Zeichenkette der Reihe nach ab. Wenn er auf den ersten Platzhalter stößt, so trägt er an dieser Stelle den Inhalt der ersten Variable nach der Zeichenkette ein. Beim zweiten Platzhalter wird dann entsprechend der Inhalt der zweiten Variable nach der Zeichenkette eingefügt. Für Mehr Details siehe Abschnitt A.1.1 auf Seite 81.

2.4 Beispiel: Gleitkommazahlen

2.4.1 Der Quellcode

```
#include <stdio.h>

/* Umwandlung von Fahrenheit in Celsius */
int main()
{
    int fahr;
    float celsius;
    int lower, upper, step;

    lower = 0;          /* untere Grenze der Temperatur */
    upper = 300;        /* obere Grenze */
    step = 20;          /* Schrittweite */

    /* Ausgabe einer Überschrift */
    printf("Fahrenheit Celsius\n");

    /* Erstellung der Tabelle */
    for(fahr=lower; fahr<=upper; fahr=fahr+step) {
        celsius = 5.0/9.0*(fahr-32.0);
        printf("%6d      %6.1f\n", fahr, celsius);
    }
    return 0;
}
```

2.4.2 Ausgabe auf dem Bildschirm

Das Programm erzeugt folgende Ausgabe auf dem Bildschirm:

```
Fahrenheit Celsius
     0      -17.8
    20       -6.7
    40        4.4
    60       15.6
    80       26.7
   100       37.8
   120       48.9
   140       60.0
   160       71.1
   180       82.2
   200       93.3
   220      104.4
   240      115.6
   260      126.7
   280      137.8
   300      148.9
```

2.4.3 Erläuterungen

Definition einer Gleitkommazahl mit *flaot*. Die Definition der Variablen erfolgt in gewohnter Weise. Auf der linken Seite wird der Datentyp notiert, das ist für einfache Gleitkommazahlen der Datentyp *float*. Rechts davon werden ein oder mehrere durch Kommata getrennte Variablennamen geschrieben. Nach dem abschließenden Semikolon ist die Definition abgeschlossen. Neben dem Datentyp *float* gibt es für Gleitkommazahlen noch die Datentypen *double* und *long double*, die im Abschnitt 3.2.5 auf Seite 30 genauer beschrieben werden.

Wiederholen von Programmstücken mit *for*. In dem Beispiel *Wiederholungen* im Abschnitt 2.3 wurde die *while*-Schleife vorgestellt. Allgemein müssen bei einer Schleife drei Dinge beachtet werden: 1. Die Schleife muss eine Anfangsbedingung haben, 2. es muss eine Abbruchbedingung, bzw. Schleifenbedingung geben, und 3. in der Schleife muss ein Wert für die Schleifenbedingung geändert werden. Die *for*-Schleife fasst diese drei Dinge in der Klammer hinter dem Befehl zusammen. Ansonsten verhält sich sich die *for*-Schleife genauso wie eine *while*-Schleife. In Tabelle 2.1 werden die beiden Schleifenarten verglichen.

***while*-Schleife**	***for*-Schleife**

fahr = lower;
while(fahr<=upper)
{
  ...
  fahr = fahr+step
}
	for(fahr=lower; fahr<=upper; fahr=fahr+step)
	{
	  ...
	}

Tabelle 2.1: Vergleich von for- und while-Schleife.

Verarbeitung von Gleitkommazahlen. Die Verarbeitung von Gleitkommazahlen erfolgt genauso wie bei ganzen Zahlen. Auch das Mischen von ganzen Zahlen mit Gleitkommazahlen ist problemlos möglich. Wichtig: in C wird als Dezimaltrennungszeichen statt dem im deutschen üblichen Komma ein Punkt verwendet.

In dem Beispielprogramm wurde hinter die Zahlen des Quotienten 5/9 jewils ein 'Komma null' angehängt. Dies ist notwendig, damit das gewünschte Zwischenergebnis von 0,555... erzielt wird. Ohne die Nachkommazahl würden die beiden Zahlen als ganze Zahlen interpretiert, und das Zwischenergebnis würde auf null abgerundet! Das Programm arbeitet stur alle Operatoren nach einer festen Reihenfolge durch, ohne von Anfang an auf den Datentyp des endgültigen Ergebnisses Rücksicht zu nehmen. Dieser Sachverhalt verdient in der C-Programmierung hohe Aufmerksamkeit und ist häufig Ursache von Fehlern.

Ausgabe von Gleitkommazahlen. Für den Datentyp *int* wird in diesem Programm wieder der Platzhalter *%d* mit einer Variation verwedet. Eine Zahl zwischen dem Prozentzeichen und dem Buchtaben *d* sorgt dafür, dass die Ausgabe mind. eine entsprechende Breite auf dem Bildschirm einnimt.

Für den Datentyp *float* wird der Platzhalter *%f* verwendet. Die Zahl 6.1 zwischen dem Prozentzeichen und dem Buchstaben *f* sorgt dafür, dass die Aus-

gabe auf dem Bildschirm mind. 6 Zeichen einnehmen, und die Zahl mit einer Nachkommastelle dargestellt werden soll. Ohne Angabe für die Anzahl der Nachkommastellen werden sechs Nachkommastellen ausgegeben. Genauere Angaben zu den Formatierungsmöglichkeiten finden Sie im Abschitt A.1.1 auf Seite 81.

Kapitel 3

Datentypen, Operatoren und Ausdrücke

3.1 Einleitung

In einem Computerprogramm werden Daten gelesen, verarbeitet und ausgegeben (Beispiel: Taschenrechner). Die Daten werden hierfür in Variablen gespeichert und bearbeitet. Die Zuweisung eines Datums in eine Variable kann in C wie folgt aussehen:

```
Wurzel2 = 1.4142;
```

Bei allen Zuweisungen steht links von Gleichheitszeichen immer eine *Variable*, hier mit dem *Variablennamen* Wurzel2. Rechts vom Gleichheitszeichen steht eine *Konstante*, hier 1.4142. Rechts vom Gleichheitszeichen kann auch eine weitere Variable oder ein *Ausdruck* stehen. Beispiel:

```
Celsius = Kelvin - 273;
```

Links vom Gleichheitszeichen steht wie immer eine *Variable*, hier mit dem Namen Celsius. Rechts vom Gleichheitszeichen steht ein *Ausdruck* bestehend aus einer weiteren *Variablen*, Kelvin, einem *Operator*, '−', und einer *Konstanten*, 273.

Ausdrücke können beliebig verschachtelt werden. Beispiel:

```
Celsius = 5*(Fahr-32)/9;
```

Allgemein ausgedrückt:

Zuweisung: *Variable = Ausdruck/Variable/Konstante*

Ausdruck: *Ausdruck/Variable/Konstante Operator Ausdruck/Variable/Konstante*

Bevor eine Variable verwendet werden kann, muss sie in einem ersten Schritt definiert werden. Dabei wird der Variablenname und der Datentyp, der in diese Variable gespeichert werden soll, festgelegt.

Im Folgenden wird auf die einzelnen Begriffe im Detail eingegangen.

3.2 Variablen

Zur Laufzeit eines Programms werden die benötigten Daten in Variablen gespeichert. In C müssen Variablen vor der ersten Verwendung definiert werden. Dieser Abschnitt beschäftigt sich mit Variablen und legt einen Fokus auf die zur Verfügung stehenden Datentypen.

3.2.1 Vereinbarungen/Definitionen

Bevor eine Variable verwendet werden kann, muss sie zuvor vereinbart/definiert werden. Die Vereinbarung erfolgt folgendermaßen, wobei die eckigen Klammern die optionalen Angaben umfassen:

Datentyp Variablenname [, *Variablenname* [...]];

Beispiel: Mit

```
int fahr, celsius;
```

werden die zwei Variablen vom Typ **int** mit den Namen fahr und celsius vereinbart, bzw. definiert. Es gilt:

- Datentypen dürfen mehrfach verwendet werden, während ein Variablenname nur einmal innerhalb eines Blocks vereinbart werden darf.
- Einer Variablen kann bei der Vereinbarung ein Startwert zugewiesen werden. Beispiel: **int** counter=0;
- Die vorgestellte Bezeichnung **const** gibt an, dass die Variable nicht verändert werden kann.
  Beispiel: **const float** pi=3.141;
- Der Versuch, eine mit **const** vereinbarte Variable zu ändern, hat undefinierte Folgen, führt aber meistens zu einem Compiler-Fehler.

3.2.2 Name einer Variablen

- Ein Variablenname besteht aus großen und kleinen Buchstaben, Ziffern und dem Tiefstrich '_'. Die deutschen Umlaute sind nicht erlaubt.
- Das erste Zeichen ist ein Buchstabe oder ein Tiefstrich (keine Ziffer).
- Große und kleine Buchstaben werden unterschieden.
- Der Name darf beliebig lang sein.
- Variablennamen werden nur anhand der ersten 31 Zeichen unterschieden.
- Reservierte Wörter wie **if**, **while** oder **break** dürfen nicht verwendet werden.

3.2.3 Elementare Datentypen

Die elementaren Datentypen sind in Tabelle 3.1 zusammengefasst. Die nachfolgenden Abschnitte gehen genauer auf die einzelnen Datentypen ein.

char	ganzzahliger Wert (ein Byte), bzw. ein Zeichen
int	ganzzahliger Wert in der auf dem Rechner 'natürlichen' Größe
float	Gleitkommazahl mit einfacher Genauigkeit.
double	Gleitkommazahl mit doppelter Genauigkeit.

Tabelle 3.1: Liste der elementaren Datentypen.

3.2.4 Ganzzahlige Datentypen

Die elementaren ganzzahligen Datentypen sind **char** und **int**. Der Datentyp **char** (engl. *character*, zu deutsch *Buchstabe*) beherbergt Platz für *einen* Buchstaben. Eine Variable vom Typ **char** kann aber auch für kleine ganze Zahlen verwendet werden.

Kleinere und größere ganzzahlige Datentypen

Die Bezeichnungen **short int** und **long int** bezeichnen einen kleinen, bzw. einen großen ganzzahligen Datentyp. In kurzer (und auch üblicher) Schreibweise kann das **int** einfach weggelassen werden. Der Datentyp **short** hat mindestens 2 Byte, **long** hat mindestens 4 Byte. Der Datentyp **int** ist vom Speicherbedarf her nicht kleiner als **short** und nicht größer als **long**. Der Speicherbedarf eines Datentyps kann mit der Funktion **sizeof**() ermittelt werden. Es gilt:

$$\textbf{sizeof}(\textbf{short}) \leq \textbf{sizeof}(\textbf{int}) \leq \textbf{sizeof}(\textbf{long})$$

Ganzzahlige Datentypen mit und ohne Vorzeichen

Den Datentypen **char**, **short**, **int** und **long** kann das Wort **signed** oder **unsigned** vorgestellt werden. Sie legen fest, ob sich der Zahlenbereich auf positive und negative oder nur auf positive Zahlen beschränkt. Die Datentypen **short**, **int** und **long** sind ohne Angabe vorzeichenbehaftet. Die Bezeichnungen **signed** bzw. **unsigned** alleine beziehen auf den Datentyp **int**.

Wertebereich der ganzzahligen Datentypen

Zur Ermittlung des Wertebereichs muss zuerst die Größe des Datentyps bekannt sein. Dafür steht die Funktion **sizeof** zur Verfügung, welche die Größe eines Datentyps in Byte ermittelt (z.B. **sizeof**(**int**)). Bezeichnet man die Anzahl der Bits mit N (1 Byte = 8 Bit), so ist der Wertebereich 2^N. Bei ganzen Zahlen ohne Vorzeichen erstreckt sich der Wertebereich von 0 bis 2^N-1. Vorzeichenbehaftete Zahlen im Zweierkomplement haben einen Wertebereich von -2^{N-1} bis $+2^{N-1}-1$.

Die tatsächlichen Grenzen der ganzzahligen Datentypen für ein System stehen in der Include-Datei 'limits.h'.

3.2.5 Datentypen für Gleitkommazahlen

Eine Gleitkommazahl besteht aus einer Mantisse und dem Vorzeichen der Mantisse, sowie dem Exponenten und dem Vorzeichen des Exponenten, z.B. +1.3E-2 für 0,013. Dabei wird das 'E' als 'mal zehn hoch' interpretiert. Wichtig: In C wird das im

deutschen Sprachraum übliche Komma für die Dezimaltrennung durch einen Punkt ersetzt.

Drei Datentypen für Gleitkommazahlen

Es stehen die Datentypen **float**, **double** und **long double** zur Verfügung. Der Speicherbedarf ist systemabhängig. Die drei Typen können für eine, zwei oder drei Speichergrößen stehen. Der Datentyp **double** ist vom Speicherbedarf her nicht kleiner als **float** und nicht größer als **long double**:

sizeof(**float**) $\leq$ **sizeof**(**double**) $\leq$ **sizeof**(**long double**)

Wertebereich der Gleitkommazahlen

Der Zahlenbereich der Gleitkommazahlen ist von der verwendeten Speichergröße, sowie der Darstellung auf Maschinenebene abhängig. Eine 4 Byte Gleitkommazahl hat üblicherweise einen Zahlenbereich von 10^{-38} bis 10^{38} im positiven wie negativem Bereich bei einer Genauigkeit von sieben Dezimalstellen und den Wert null.

Die tatsächlichen Grenzen der Datentypen für Gleitkommazahlen auf einem System stehen in der Include-Datei '**float**.h'.

3.2.6 Zusammenfassung der wichtigsten Datentypen

Datentyp	Speicherbedarf	Kommentar
char	1 Byte	ganze Zahl oder ein Zeichen
short int	mind. 2 Byte	ganze Zahl
short	**short** $\leq$ **int**	
int	**short** $\leq$ **int** $\leq$ **long**	ganze Zahl
long int	mind. 4 Byte	ganze Zahl
long	**int** $\leq$ **long**	
float	**float** $\leq$ **double**	Gleitkommazahl
double	**float** $\leq$ **double** $\leq$ **long double**	Gleitkommazahl
long double	**double** $\leq$ **long double**	Gleitkommazahl

3.2.7 Zeichenketten

Zeichenketten sind eine Aneinanderreihung (Verkettung) von Zeichen (**char**). Der Speicherbedarf hängt von der Länge der zu speichernden Zeichenketten ab. Da in C die Zeichenketten mit ASCII-Code null enden, ist die Anzahl der benötigten Bytes ist um eins größer als die Anzahl der zu speichernden Zeichen (siehe Abschnitt 3.3.4).

3.2.8 Die vier Eigenschaften von Variablen

Jede Variable in C hat vier Eigenschaften:

Datentyp. Der Datentyp gibt an, welche Art von Daten in die Variable gespeichert werden sollen, siehe Abschnitt 3.2.3 und folgende.

Variablenname. Um eine Variable eindeutig ansprechen zu können wird ihr ein Variablenname zugewiesen.

Wert. Eine Variable hat einen Wert. Das ist der eigentliche Sinn und Zweck einer Variablen. Der Wert ist zu Beginn undefiniert.

Speicherort/Adresse. Eine Variable wird an einer Stelle im Speicher abgelegt. Dieser Speicherort ist die Adresse der Variable (ähnlich wie eine Postadresse) und kann in C besonders verwendet werden.

3.3 Konstanten

3.3.1 Ganze Zahlen

Es gelten folgende Regeln:

- Eine einfache ganze Zahl wird als *int* interpretiert.
  Beispiel: 1234
- Der Typ *long* wird durch ein 'l' oder 'L' direkt hinter der Zahl erzeugt.
  Beispiel: 1234L
- Eine vorzeichenlose Zahl wird durch ein 'u' oder 'U' angedeutet.
  Beispiel 1234u
- Die Endung 'ul' bezeichnet den Typ *unsigned long*. Reihenfolge und Groß- und Kleinschreibung der beiden Buchstaben haben keinen Einfluss.
  Beispiel: 1234ul
- Beginnt eine Zahl mit einer Null, so wird sie als Oktalzahl interpretiert.
  Beispiel: 045 (ergibt dezimal: 37)
- Eine hexadezimale Zahl wird durch eine Null gefolgt von einem 'x' oder 'X' eingeleitet.
  Beispiel: 0x3f (ergibt dezimal: 63)
- Kombinationen sind möglich.
  Beispiel: 0XFUL (ergibt die dezimale Zahl 15 vom Typ *unsigned long*).

3.3.2 Gleitkommazahlen

In C werden Gleitkommazahlen durch Mantisse und Exponenten dargestellt. Es gilt:

- Gleitkommakonstanten enthalten einen Punkt und/oder einen Exponenten.
  Beispiel: 1.234e5 (steht für 123400), 6.54 (steht für 6,54), 2e−3 (steht für 0,002)
- Ohne nachgestellten Buchstaben wird die Zahl als double interpretiert.
  Beispiel: 3.141 und 1.6e−19 sind vom Typ *double*.
- Die Endung 'f' oder 'F' deutet den Typ float an.
  Beispiel: 2.7183f, 1e−6f und 12F sind vom Typ *float*.
- Die Endung 'l' oder 'L' deutet den Typ long double an.
  Beispiel: 0.142857142857142857L und 1e−100L sind vom Typ *long double*.

3.3.3 Einzelne Zeichen

- Einzelne Zeichen werden in einfache Anführungszeichen gesetzt.
  Beispiel: 'b'

- Der Wert eines Zeichens ist der numerische Wert des auf dem System verwendeten Zeichensatzes.
  Beispiel: Das Zeichen '0' (Null) hat im ASCII-Zeichensatz den Code 48, siehe Tabelle B.1 auf Seite 92.

- Ein Zeichen wird wie eine Zahl behandelt, die zum Rechnen verwendet werden kann.
  Beispiel: 'A'+1 hat im ASCII-Zeichensatz das Ergebnis 'B'. (Im ASCII-Zeichensatz hat 'A' den Code 65 und 'B' den Code 66.) Siehe Tabelle B.1 auf Seite 92.

- Einige Sonderzeichen werden über eine Zeichenfolge beginnend mit einem Gegenstrich angedeutet. (Die Zeichenfolge steht dabei für *ein* Zeichen.)
  Beispiel: Das Zeichen '\n' steht für einen Zeilenvorschub.

- Der Code eines Zeichens kann oktal direkt hinter dem Gegenstrich angegeben werden.
  Beispiel: '\101' ist identisch zu 'A'

- Der Code eines Zeichens kann hexadezimal (sedezimal) hinter dem Gegenstrich gefolgt von einem 'x' angegeben werden.
  Beispiel: '\x41' entspricht dem 'A'

Die zur Verfügung stehenden Sonderzeichen sind in Tabelle 3.2 zusammengefasst.

\n Zeilenvorschub	\r Wagenrücklauf
\t Tabulator	\v Vertikaltabulator
\b backspace	\f Seitenvorschub
\' Einfachanführungszeichen	\" Doppelanführungszeichen
\a Klingelzeichen	\? Fragezeichen
\ooo ASCII-Code (oktal)	\xhh ASCII-Code (hexadezimal)
\\ Gegenstrich	

Tabelle 3.2: Die Sonderzeichen im Überblick

3.3.4 Zeichenketten (Strings)

- Eine Zeichenkette wird durch Doppelanführungszeichen angedeutet und begrenzt. Sie kann eine beliebige Anzahl an Zeichen umfassen.
  Beispiel: "Ich␣bin␣der␣ich␣bin." (Hinweis: In diesem Skript wird zur Hervorhebung von Leerzeichen ein eigenes Sonderzeichen verwendet: "␣". Verwenden Sie in Ihrem Programm einfach die Leertaste.)

- Auch die Länge null ist erlaubt.
  Beispiel: ""

- Es gelten dieselben Sonderzeichen wie bei Zeichenkonstanten, siehe Tabelle 3.2.
  Beispiel: "Ich␣bin␣der␣\"ich␣bin\"." wird zu: Ich bin der "ich␣bin".

- Aufeinander folgende Zeichenketten werden als eine Zeichenkette interpretiert. Dieser Mechanismus kann sinnvol eingesetzt werden, um eine Zeichenkette auf mehrere Zeilen zu verteilen.
  Beispiel: Die folgenden zwei Anweisungen erzeugen exakt die selbe Ausgabe:
  ```
  printf("Ich␣bin␣der␣"
      "ich␣bin.\n");
  printf("Ich␣bin␣der␣ich␣bin.\n");"
  ```

- Eine Zeichenkette ist ein Vektor (eine 'Kette') von Zeichen.

- Das Ende einer Zeichenkette wird intern durch eine nachgestellte null ('\0') begrenzt. Daher benötigt eine Zeichenkette im Speicher ein Byte mehr als Anzahl der Zeichen.

- Man beachte den Unterschied zwischen 'x' und "x". Hinter 'x' verbirgt sich der Code des Buchstabens 'x', während "x" eine Zeichenkette mit dem einen Buchstaben 'x' gefolgt von einer null ist.

3.4 Operatoren

Ein Operator verarbeitet Operanden. Beispiel: der Ausdruck 2+3 hat den Operator + und die Operanden 2 und 3.

Es gibt *unäre*, *binäre* und *ternäre* Operatoren. *Unäre* Operatoren erwarten einen Operanden. Beispiel: Im Ausdruck „-a“ erwartet das negative Vorzeichen *einen* Operanden und liefert als Ergebnis den Wert mit umgekehrten Vorzeichen zurück. *Binäre* Operatoren erwarten zwei Operanden. Beispiel: Im Ausdruck „a-b“ erwartet der Subtraktion-Operator *zwei* Operanden und liefert die Differenz zurück. *Ternäre* Operatoren erwarten *drei Operanden*. Beispiel: Der bedingte Ausdruck „a¡b?a:b“ gibt von a und b den kleineren Wert zurück, siehe Abschnitt 3.4.7.

Die Unterscheidung zwischen gleichnamigen Operatoren ergibt sich aus dem Kontext. Beispiel: $3*-4$ entspricht $3*(-4)$, Ergebnis -12.

3.4.1 Arithmetische Operatoren

- Es gibt fünf arithmetische Operatoren. Die fünf Operatoren sind die vier Grundrechenarten dargestellt durch die Zeichen +, −, * und /, sowie die Restdivision angedeutet durch das Zeichen %.
  Beispiel: Die Restdivision 17 % 5 hat das Ergebnis 2.

- Die vier Grundrechenarten können auf alle Ganzzahl- und Gleitkommatypen angewendet werden. Das Verhalten bei Bereichsüberschreitung ist unterschiedlich. Division durch null führt zu einem Fehler und muss bei der Programmierung unbedingt berücksichtigt werden. Der Operator % kann nur auf ganzzahlige Datentypen angewendet werden. Bei negativen Zahlen ist das Ergebnis maschinenabhängig und damit undefiniert.

- Es gilt Punkt- vor Strichrechnung, bei gleichem Rang wird von links nach rechts gerechnet. Der Operator % gehört zur Punktrechnung.

- Die unären Operatoren + und − (Vorzeichen einer Zahl) haben Vorrang gegenüber der Punktrechnung.

3.4.2 Vergleichsoperatoren

Die sechs Vergleichsoperatoren sind in Tabelle 3.3 zusammengefasst.

==	gleich wie	<	kleiner als	<=	kleiner gleich als
!=	ungleich wie	>	größer als	>=	größer gleich als

Tabelle 3.3: Die sechs Vergleichsoperatoren.

- Das Ergebnis einer Vergleichsoperation ergibt *wahr* oder *nicht wahr*.
  Beispiel: 3==4 ergibt *nicht wahr* während 5<8 *wahr* ergibt.

- Die beiden Ergebnisse *wahr* und *nicht wahr* können auch als Zahlen mit den entsprechenden Werten eins und null interpretiert werden.

- Arithmetische Operatoren (siehe Abschnitt 3.4.1) haben einen höheren Rang als die Vergleichsoperatoren.
  Beispiel: counter<max−1 wird wie counter<(max−1) bewertet.

- Die Operatoren == und != haben einen geringeren Rang als die anderen Vergleichsoperatoren.
  Beispiel: 3<4==2>1 wird bewertet wie (3<4)==(2>1).

3.4.3 Verknüpfungs-/Logikoperatoren

Die Vergleichsoperatoren liefern logische Ergebnisse (nur *wahr* oder *nicht wahr*) die mit den Verknüpfungsoperatoren logisch verknüpft werden können. Es gibt zwei Verknüpfungsoperatoren: die Und-Verknüpfung dargestellt mit && und die Oder-Verknüpfung dargestellt mit ||.

Die Verknüpfungsoperatoren erwarten zwei logische Werte und liefern einen logischen Wert zurück. Die Wahrheitstabelle der beiden Operatoren ist in Tabelle 3.4 zusammengefasst.

linker Operand	**rechter Operand**	**Oder-Verknüpfung**	**Und-Verknüpfung**
nicht wahr	nicht wahr	nicht wahr	nicht wahr
nicht wahr	wahr	wahr	nicht wahr
wahr	nicht wahr	wahr	nicht wahr
wahr	wahr	wahr	wahr

Tabelle 3.4: Wahreitstabelle der Verknüpfungsoperatoren

- Die Verknüpfungsoperatoren können auch auf Zahlen angewendet werden. Der Zahlenwert null wird als *nicht wahr* interpretiert, während alle Zahlen ungleich null als *wahr* bewertet werden.
- Die Oder-Verknüpfung hat einen geringeren Rang als die Und-Verknüpfung.
- Die Verknüpfungsoperatoren haben einen geringeren Rang als die Vergleichsoperatoren.
- Ausdrücke, die mit Vergleichsoperatoren verknüpft sind, werden solange von links nach rechts bearbeitet, bis das Ergebnis der Verknüpfung feststeht.
  Beispiel: Bei `2>3&&3*7==21` wird die Multiplikation 3*7 und der Vergleich mit 21 nicht mehr durchgeführt, da mit dem Vergleich 2>3 und der Und-Verknüpfung bereits feststeht, dass das Ergebnis des gesamten Ausdrucks *nicht wahr* ist.

3.4.4 Negationsoperator

- Der unäre Negationsoperator wird durch das Ausrufungszeichen `!` dargestellt und negiert den logischen Wert eines Ausdrucks.
  Beispiel: `!(2==3)` hat das Ergebnis *wahr*.
- Auf einen Zahlenwert angewendet wird eine 0 (*nicht wahr*) zu einer 1 (*wahr*). Alle Zahlenwerte ungleich null ergeben durch Negation den Wert null.
  Beispiel: `!0` ergibt *wahr* (Zahlenwert 1), `!1` und `!7` ergeben *nicht wahr* (Zahlenwert 0).

3.4.5 Inkrement und Dekrement

Es gibt zwei unäre Operatoren zum Inkrementieren und Dekrementieren von ganzen Zahlen.

- Der Inkrementoperator, dargestellt durch `++`, erhöht eine ganze Zahl um eins.
  Beispiel: `i++;`
- Der Dekrementoperator, durch `--` dargestellt, erniedrigt eine ganze Zahl um eins.
  Beispiel: `i--;`
- In der Präfixnotation wird der Operator vor den Operanden geschrieben, und bewirkt, dass erst der Operator angewendet wird, und dann das Ergebnis zurückgegeben wird.
  Beispiel: Hat `b` den Wert 2 so weist der Ausdruck `a=++b;` der Variable `a` den Wert 3 zu. Die Variable `b` wird um eins auf 3 erhöht.
- In der Postfixnotation wird der Operator hinter den Operanden geschrieben, so dass die Variable erst verändert wird nachdem sie weiter verarbeitet wurde.
  Beispiel: Hat b den Wert 2, so weist der Ausdruck `a=b++;` der Variable `a` den Wert 2 zu. Die Variable `b` wird um eins auf 3 erhöht.
- Die Operatoren können nur auf Variablen angewandt werden. Ausdrücke wie `7++` oder `(a+b)++` sind nicht erlaubt.

3.4.6 Bitmanipulation

Für diesen Abschnitt ist das Verständnis von dualen (binären) und hexadezimalen (sedezimalen) Zahlen Voraussetzung (siehe Abschnitt 1.3). Zur Bitmanipulation stehen sechs Operatoren zur Verfügung, siehe Tabelle 3.5.

~	Bit-Komplement (unärer Operator)
&	bitweise UND-Verknüpfung
|	bitweise ODER-Verknüpfung
^	bitweise Exklusiv-ODER-Verknüpfung
<<	Bit-Verschiebung nach links
>>	Bit-Verschiebung nach rechts

Tabelle 3.5: Die sechs Operatoren zur Bitmanipulation

Die Operatoren zur Bitmanipulation können auf alle ganzzahligen Datentypen angewendet werden (**char**, **short**, **int** und **long**, **signed** wie **unsigned**).

Bit-Komplement

Das Bit-Komplement ist ein unärer Operator, der als Ergebnis seinen Operanden mit umgekippten Bits liefert. Beispiel:

```
unsigned char c1, c2;
c1 = 0x0f;
c2 = ~c1;
```

In diesem Programmstück wird c1 der hexadezimale Wert 0F (dual: 00001111) zugewiesen. Das Bit-Komplement kippt alle Bits um, so dass sich in c2 ein Ergebnis von hexadezimal F0 (dual: 11110000) ergibt.

Die bitweise UND, ODER und EXKLUSIV ODER Verknüpfung

Werden zwei Bits mit bitweisen UND-, ODER- oder der Exklusiv-ODER-Operoren verknüpft, so ergeben sich folgende Ergebnisse gemäß Tabelle 3.6.

Bit 1	Bit 2	UND	ODER	EXKLUSIV ODER
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

Tabelle 3.6: Wahrheitstabelle der bitweisen UND-, ODER- und EXKLUSIV ODER-Operatoren.

Beispiel: Nach dem folgenden Programmstück haben die Variablen c, d und e die Werte 0x18 (dual: 0001 1000), 0x7E (dual: 0111 1110) und 0x66 (dual: 0110 0110). Siehe auch die nachfolgende Tabelle.

```
int a=0x5C, b=0x3A, c, d, e;
c = a&b;
d = a|b;
e = a^b;
```

Variable	Ausdruck	hexadezimal	dual
a	-	$5C_{16}$	$0101\,1100_2$
b	-	$3A_{16}$	$0011\,1010_2$
c	a&b	18_{16}	$0001\,1000_2$
d	a|b	$7E_{16}$	$0111\,1110_2$
e	a^b	66_{16}	$0110\,0110_2$

- Mit der UND-Verknüpfung können gezielt Bits auf null gesetzt werden.
  Beispiel: Ist `c` vom Typ **char**, so werden bei der Zuweisung `c=c&0x0f;` die ersten vier der acht Bits von `c` auf null gesetzt. Die hinteren vier Bits bleiben unverändert.

- Mit der ODER-Verknüpfung können gezielt Bits auf eins gesetzt werden.
  Beispiel: Ist `c` vom Typ **char**, so werden bei der Zuweisung `c=c|0xf0;` die ersten vier Bits auf eins gesetzt. Die anderen Bits bleiben unverändert.

- mit der UND-Verknüpfung können gezielt einzelne Bits abgefragt werden.
  Beispiel: Der Ausdruck `c&8` ist *wahr*, wenn das Bit 3 von `c` gesetzt ist.

- Mit der EXKLUSIV-ODER-Verknüpfung können gezielt einzelne Bits negiert werden.
  Beispiel: Mit `c=c^6` werden Bit 1 und Bit 2 von `c` negiert.

- Mit der EXKLUSIV-ODER-Verknüpfung können die sich unterscheidenen Bits von zwei Werten ermittelt werden.
  Beispiel: Nach der Programmzeile `a=b^c` sind in `a` die Bits gesetzt, bei denen sich die Variablen `b` und `c` unterscheiden.

Bit-Verschiebung

- Die Operatoren zur Bit-Verschiebung verschieben die dualen Ziffern um eine gewünschte Anzahl von Ziffern.
  Beispiel: Der Ausdruck `0x02<<3` liefert das Ergebnis 0x10 (dual: 00010000).

- Eine Verschiebung um N Stellen nach links kommt einer Multiplikation mit 2^N gleich.
  Beispiel: `3<<5` ergibt den Wert $3 \cdot 2^5 = 96$.

- Bei einer Verschiebung nach links werden die neuen Stellen mit null aufgefüllt.

- Werden die Bits eines vorzeichenlosen Werts (Typ **unsigned**) nach rechts verschoben so wird von links mit nullen aufgefüllt.

- Werden die Bits eines vorzeichenbehafteten Werts (Typ **signed**) nach rechts geschoben, so wird maschinenabhängig entweder mit dem Vorzeichenbit (*arithmetic shift*) oder mit null (*logical shift*) aufgefüllt.

3.4.7 Bedingter Ausdruck

Die bedingte Anweisung

```
if(a>b) z = a;
else z = b;
```

(Ermittlung des Maximalwertes) kann verkürzt wie folgt ausgedrückt werden:

```
z = (a>b) ? a : b;
```

(Wenn a größer als b ist, soll z den Wert von a erhalten, andernfalls den Wert von b.) Allgemein ausgedrückt:

Ausdruck1 ? *Ausdruck2* : *Ausdruck3*

Als erstes wird Ausdr1 ausgewertet. Ist das Ergebnis ungleich null, also *wahr*, so wird Ausdr2 bewertet und das Ergebnis zurückgegeben. Ist das Ergebnis von Ausdr1 gleich null, so wird Ausdr3 bewertet und dessen Ergebnis zurückgegeben. Der jeweils andere Ausdruck wird nicht ausgewertet. Es gelten die gleichen Regeln für die Typumwandlung wie bei binären Operatoren (siehe Abschnitt 2.6.1).

Beispiel: Das folgende Programm berechnet die Quadratzahlen von null bis i_{max}. Dabei soll nach zehn Zahlen und am Ende ein Zeilenvorschub, ansonsten nach jeder Zahl ein Leerzeichen eingefügt werden.

```
for(i=0; i<=i_max; i++) {
    printf("%d", i*i);
    if(i%10==9 || i==i_max) printf("\n");
    else printf(" ");
}
```

Die drei printf-Befehle können mit einem bedingten Ausdruck zusammengefasst werden:

```
for(i=0; i<=i_max; i++)
    printf("%d%c", i*i, (i%10==9 || i==i_max) ? '\n' : ' ');
```

Eine anderes Beispiel zur korrekten Schreibweise von Einzahl/Mehrzahl:

```
printf("Sie haben %d Teil%s.\n", n, (n==1) ? "" : "e");
```

Der bedingte Operator hat einen sehr geringen Rang, daher könnten die Klammen um den Ausdruck vor dem Fragezeichen auch weggelassen werden. Aber als Teil eines größeren Ausdrucks muss der Bedingte Ausdruck in den meisten Fällen in Klammern gesetzt werden.

3.5 Typumwandlung

3.5.1 Typumwandlung bei binären Operatoren

Wird ein Operator auf zwei unterschiedliche Datentypen angewendet, so stellt sich die Frage, welchen Typ das Ergebnis hat. In C werden vor Anwendung eines Operators die Typen beider Operanden verglichen und angepasst. Dafür werden die Datentypen in einer Rangfolge betrachtet, siehe Tabelle 3.7. Befindet sich einer der Operanden auf einem niedrigeren Rang, so wird er vor Anwendung der Operators auf den Rang des anderen Operanden umgewandelt.

Beispiel: Im Ausdruck 7−3ul ist der Minuend (linker Operand) vom Typ **int** und der Subtrahend (rechter Operand) vom Typ **unsigned long**. Vor der Durchführung der Subtraktion wird der Minuend in den Typ **unsigned long** umgewandelt. Das Ergebnis hat ebenfalls den Datentyp **unsigned long**.

Die Datentypen **char** und **short** werden vor Anwendung der Operatoren mindestens in den Typ **int** umgewandelt.

Rang	Datentyp
1	**long double**
2	**double**
3	**float**
4	**unsigned long**
5*	**long**
6*	**unsigned int**
7	**int**

Tabelle 3.7: Rangordung bei der Typumwandlung. *Siehe Anmerkung im Text.

Es gibt einen Sonderfall: Wenn ein Operand vom Typ **unsigned int**, und der andere vom Typ **long** ist, so gibt es zwei Möglichkeiten: Umfasst der positive Teil von **long** den gesamten Zahlenbereich von **unsigned int**, so wird der Typ **unsigned int** in **long** umgewandelt. Andernfalls (wenn **int** und **long** den gleichen Zahlenbereich haben) werden beide zum Typ **unsigned long**.

3.5.2 Typumwandlung bei Zuweisungen

Bei einer Zuweisung wird der Datentyp der rechten Seite in den Datentyp der linken Seite umgewandelt. Wird der Zahlenbereich dadurch eingeschränkt, so kann es bei der Übersetzung zu einer Warnung kommen. Zwei Beispiele:

```
int i;
char c=65;

i = c;
c = i;
```

```
int i=12345;
char c;

c = i;
i = c;
```

Bei dem linken Beispiel wird in der Variable `c` am Ende immer noch der Wert 65 stehen. Beim rechten Code dagegen wird sich der Inhalt von `i` ändern. In beiden Fällen kann es bei der Zeile `c = i;` zu einer Warnung kommen.

Bei der Reduzierung einer Gleitkommazahl in eine ganze Zahl wird der Nachkommateil abgeschnitten.

Bei Reduzierung innerhalb der Gleitkommatypen ist das Verhalten maschinen-, bzw. übersetzerabhängig. Entweder wird mathematisch korrekt gerundet, oder es wird einfach abgeschnitten.

3.5.3 Explizite Typumwandlung (cast)

Mit einem *cast* (zu deutsch „*Gipsverband*“) kann ein Datentyp explizit umgewandelt werden. Der gewünschte Typ wird in runden Klammern vorgestellt. Beispiel:

```
int i=65;
char c;

c = (char)i;
```

In diesem Beispiel wird die Warnung unterdrückt, die sonst einige Übersetzer bei der Zuweisung `c = i;` erzeugen würden.

3.6 Zuweisungen und Ausdrücke

3.6.1 Verkürzte Darstellung von Zuweisungen

In der Zeile

```
i = i+3;
```

steckt ein Ausdruck (i+3) und eine Zuweisung (i = Ausdruck). Dies kann in C verkürzt dargestellt werden als:

```
i += 3;
```

Diese Schreibweise erspart Redundanzen und entspricht auch mehr unserem Sprachgebrauch: „Erhöhe i um drei.“ statt „Weise i den Wert i plus drei zu.“

Für die folgenden binären Operatoren *op* sind entsprechende Zuweisungsoperatoren *op*= erlaubt: +, −, ∗, /, %, &, |, ˆ, << und >>. Allgemein gilt:

Variable Operator= Ausdruck

ist äquivalent zu

Variable = Variable Operator (*Ausdruck*)

Die Klammer macht deutlich, dass z.B.

```
a *= b+2;
```

nicht als

```
a = a*b+2;   /* falsche Interpretation */
```

sondern als

```
a = a*(b+2); /* richtige Interpretation */
```

bewertet wird.

3.6.2 Ergebnis einer Zuweisung

Eine Zuweisung ist zugleich ein Ausdruck, der den zugewiesenen Wert zurück gibt. Z.B. wird in

```
a = (b=3)*4;
```

der Variable b in der Klammer der Wert drei zugewiesen. Danach wird der neue Wert von b multipliziert mit vier der Variable a zugewiesen, so dass nach dieser Zeile a den Wert 12 hat.

3.7 Rangfolge der Operatoren

In der folgenden Tabelle sind alle Operatoren von standard C mit ihrer Rangfolge zusammengefasst. Die letzte Spalte gibt die Reihenfolge der Verarbeitung eines Ranges innerhalb eines Ausdrucks an. So wird z.B. im Ausdruck $3 * 4/6$ erst die Multiplikation, und dann die Division durchgeführt. (Es ist nicht die Reihenfolge der Operatoren in der Tabelle gemeint.)

Rang	Operatoren	Reihenfolge
1	() [] –> .	von links nach rechts
2	! ~ ++ −− + − * & (type) **sizeof**	*von rechts nach links*
3	* / %	von links nach rechts
4	+ −	von links nach rechts
5	<< >>	von links nach rechts
6	< <= > >=	von links nach rechts
7	== !=	von links nach rechts
8	&	von links nach rechts
9	^	von links nach rechts
10	|	von links nach rechts
11	&&	von links nach rechts
12	||	von links nach rechts
13	?:	*von rechts nach links*
14	= += −= *= /= %= &= ^= |= <<= >>=	*von rechts nach links*
15	,	von links nach rechts

Bei den Operatoren +, −, * und & im Rang 2 handelt es sich um die unären Varianten (positives und negatives Vorzeichen, Verweis- und Adressoperator). Die Operatoren '[]', '–>' und '.' in Rang 1, die unären Operatoren '&' und '*' in Rang 2, sowie das Komma in Rang 15 werden zu einem späteren Zeitpunkt behandelt.

3.8 Aufgaben

Aufgabe 3.1: Aus welchen Zeichen dürfen Variablennamen bestehen?

Aufgabe 3.2: Nennen Sie fünf Beispiele für Variablennamen, die trotz erlaubter Zeichen nicht erlaubt sind.

Aufgabe 3.3: Definieren Sie eine Variable vom Typ **int** mit Namen Zahl.

Aufgabe 3.4: Definieren Sie eine Variable für große ganze Zahlen ohne Vorzeichen mit Namen Nummer.

Aufgabe 3.5: Definieren Sie eine Gleitkommazahl.

Aufgabe 3.6: Nennen Sie alle Gleitkommatypen.

Aufgabe 3.7: Geben Sie Dezimalwert und Typ folgender Konstanten an:

a) 12L b) 012 c) 0x12 d) 0XaL e) 01U f) 0xBALU

Aufgabe 3.8: Was bedeuten folgende Sonderzeichen?

a) \n b) \r c) \t d) \b e) \a f) \\

Aufgabe 3.9: Was ergeben die folgenden Ausdrücke?

a) $2 + 3 * 4$ b) $(5 - 1)/2$ c) $5/2$ d) $6\%4$ e) $2 + 4\%3$
f) $2/3 * 5$ g) $2 * 1.5$ h) $3/2 * 2.4$ i) $3 * 2.4/2$

Aufgabe 3.10: Was ergeben die folgenden Ausdrücke?

a) $1 < 2$ b) $2! = 3$ c) $2 >= 2$
d) $2 > 1\&\&3 == 4$ e) $5! = 6||1$ f) $2\&\&!(2 == 2)$

Aufgabe 3.11: Für jede der Aufgaben a) bis f) gilt: x und y sind vom Typ **int** und haben den Wert 1. Was ergeben die folgenden Ausdrücke und welche Werte haben danach die Variablen x und y?

a) $x <= --y$ b) $x <= y--$ c) $x++||y++$
d) $--x\&\&++y$ e) $(x++) + (++y)$ f) $(x--) - (--y)$

Aufgabe 3.12: Was ergeben die folgenden Ausdrücke?

a) 7&3 b) 7|3 c) 1<<3 d) 14>>2 e) 3^5 f) ~12345^12345

Aufgabe 3.13: Was ergeben die folgenden Ausdrücke?

a) 1?2:3 b) ~0 == -1?-2:-3 c) -1^~0?1:2

Kapitel 4

Kontrollstrukturen

Ein Programm besteht aus Anweisungen, die vom Computer abgearbeitet werden. Nun ist es aber sinnvoll, dass einige Anweisungen nur bedingt und andere vielleicht mehrfach ausgeführt werden. Dafür stehen in C einige Kontrollstrukturen zur Verfügung, die in diesem Abschnitt behandelt werden.

Zur Einführung der Kontrollstrukturen werden zur Veranschaulichung Aktivitätsdiagramme (Element der *Unified Modelling Language*, UML 2.x) verwendet. Im Anhang C findet sich noch einmal eine „unvollständige“ Einführung in Aktivitätsdiagramme, wo auch auf typische Fehler hingewiesen wird.

4.1 Anweisungen und Blöcke

Anweisungen können in C beliebig verkettet werden. Eine Folge von Anweisungen ist in Abbildung 4.1 dargestellt.

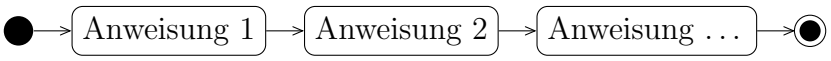


Abbildung 4.1: Aktivitätsdiagramm für eine Folge von Anweisungen.

In C können mehrere Anweisungen durch geschweifte Klammern zu einem Block zusammengefasst werden. Beispiel:

```
for(i=1; i<=10; i++) {
    q = i*i;
    printf("%d zum Quadrat ist: %d\n", i, q);
}
```

Hier umfassen die geschweiften Klammern zwei Anweisungen. Ohne die Klammern würde sich die for-Schleife nur auf die erste der beiden Anweisungen beziehen.

Alle im Folgenden aufgeführten Verzweigungen und Schleifen beziehen sich ohne geschweifte Klammern nur auf eine einzelne Anweisungen (Ausnahme: Mehrfachverzweigung mit **switch**, siehe Abschnitt 4.2.4). Sollen z.B. in einer Schleife mehrere Anweisungen bearbeitet werden, so müssen diese mit geschweiften Klammern zusammengefasst werden.

Hinweis: Bei Aktivitätsdiagrammen sprechen wir von *Aktionen* die in abgerundeten Rechtecken dargestellt werden. Dabei kann eine Aktion bereits mehrere Anweisungen enthalten.

4.2 Verzweigungen

4.2.1 Bedingte Verarbeitung (if-Verzweigung)

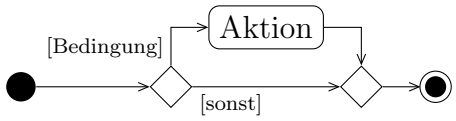


Abbildung 4.2: Aktivitätsdiagramm einer bedingten Verarbeitung (if).

Eine bedingte Verarbeitung kann durch ein Aktivitätsdiagramm wie in Abbildung 4.2 dargestellt werden. Wenn die Bedingung *wahr* ergibt, so wird der Block *Aktion* ausgeführt. Ergibt die Bedingung *nicht wahr*, so wird der Block umgangen. Die Syntax in C lautet:

if(*Bedingung*) *Aktion*

Beispiel:

```
if(i%10==9) printf("\n");    /* Zeilenvorschub bei jedem 10. i */
```

4.2.2 Einfache Alternative (if-else-Verzweigung)

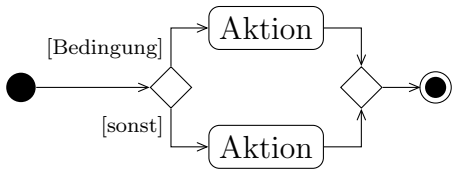


Abbildung 4.3: Aktivitätsdiagramm einer einfachen Alternative (if-else).

Bei einer einfachen Alternative gibt es für beide Fälle der Bedingung die Verarbeitung eiuner Aktion, siehe Abbildung 4.3. Die Syntax lautet:

if(*Bedingung*) *Aktion*
else *Aktion*

Beispiel:

```
if(Geschlecht==weiblich) printf("weiblich");
else printf("männlich");
```

Verzweigungen können wie alle Kontrollstrukturen beliebig verschachtelt werden. Beispiel:

```
if(Monat==Februar)
    if(Jahr==Schaltjahr) Tage = 29;
    else Tage = 28;
```

Das aufgeführte Beispiel macht eine wichtige Frage deutlich: Zu welchem *if* gehört das *else*? Die Antwort lautet: Es gehört zu dem letzten *if*, bei dem ein *else* noch möglich ist. Um eine falsche Interpretation zu vermeiden, ist das korrekte Einrücken sehr hilfreich. Das *else* sollte immer genau soweit wie das dazugehörige *if* eingerückt werden. Aber Vorsicht, der Übersetzer kümmert sich nicht um die Einrückung. Wenn falsch eingerückt wird, so ist das noch verwirrender, als wenn überhaupt nicht eingerückt wird.

Das Beste ist, in unsicheren Situationen geschweifte Klammern zu verwenden. Das letzte Beispiel könnte auch wie folgt aussehen:

```
if(Monat==Februar) {
    if(Jahr==Schaltjahr) Tage = 29;
    else Tage = 28;
}
```

Mit diesem Klammerpaar machen Sie dem Leser und dem Übersetzer unmissverständlich klar, zu welchem *if* das *else* gehören soll.

4.2.3 Verkettete Verzweigungen (else-if-Verzweigung)

Der *if*-Befehl bietet eine besondere Möglichkeit: Es können mehrere *if*-Verzweigungen verkettet werden. Abbildung 4.4 zeigt eine dreifach verkettete *if*-Verzweigung.

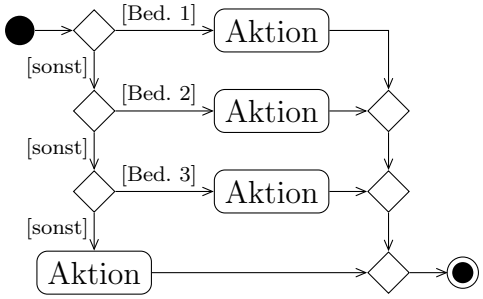


Abbildung 4.4: Aktivitätsdiagramm einer verketteten Verzweigung (else-if).

Zum Beispiel könnte der Besitzer eines Juwelierladens mit seinen Kunden folgendermaßen verfahren, vorausgesetzt, er kennt dessen Jahreseinkommen E, siehe Abbildung 4.5. Das dazugehörige Programm in C könnte wie folgt aussehen:

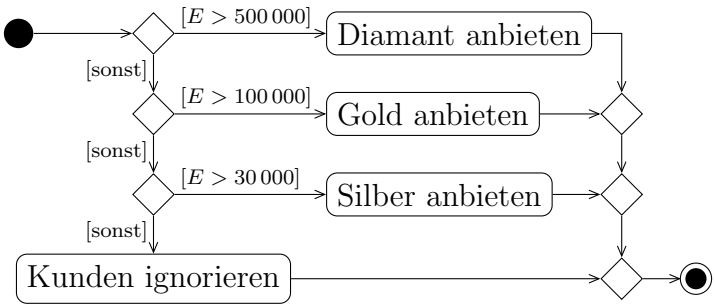


Abbildung 4.5: Beispiel einer verketteten Verzweigung.

```
if(E>500000) printf("Darf es ein Diamant für Sie sein?\n");
else if(E>100000) printf("Ich habe einen Goldring für Sie.\n");
else if(E>30000) printf("Kaufen Sie Silber!\n");
else printf("Ich habe gerade leider keine Zeit für Sie.\n");
```

4.2.4 Mehrfache Alternative (switch-case-Verzweigung)

Es gibt Situationen, in denen man sich zwischen mehr als zwei gleichberechtigten Möglichkeiten entscheiden muss. Wenn zum Beispiel die vier Jahreszeiten mit den Ziffern 1 bis 4 nummeriert sind, so muss für eine Ausgabe in Textform zwischen vier gleichberechtigten Möglichkeiten unterschieden werden. In C gibt es hierfür die *switch*-Verzweigung. Die allgemeine Syntax lautet:

switch(*Variable*) {
case *Konstante1*: *Aktion* [**break**;]
[**case** *Konstante2*: *Aktion* [**break**;]]
[**case** *Konstante3*: *Aktion* [**break**;]]
[...]
[**default**: *Aktion* [**break**;]]
}

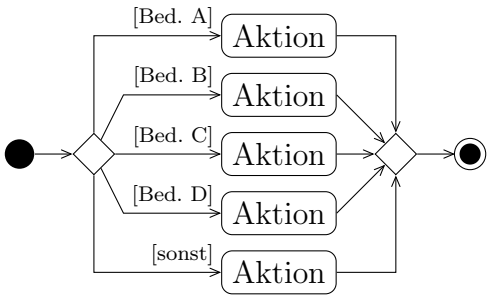


Abbildung 4.6: Mehrfach-Verzweigung

Das Beispiel mit den vier Jahreszeiten würde folgendermaßen implementiert:

```
switch(Jahrenzeit) {
case 1: printf("Frühling"); break;
case 2: printf("Sommer"); break;
case 3: printf("Herbst"); break;
case 4: printf("Winter"); break;
default: printf("Unbekannte Jahreszeit"); break;
}
```

Die einzelnen *case*-Zeilen sind wie Sprungmarken zu betrachten. Hat die Verzweigungsbedingung (im Beispiel Jahreszeit) den Wert, der hinter dem *case* steht, so werden die folgenden Anweisungen bis zum nächsten *break*-Befehl ausgeführt. Die Zeile mit dem Schlüsselwort *default* wird ausgeführt, wenn keines der anderen Bedingungen zutrifft.

Die break-Befehle können gezielt verwendet werden. Sollen z.B. die Anzahl der Tage eines Monats ermittelt werden, sähe das in etwa so aus:

```
switch(Monat) {
case 4:                          /* April */
case 6:                          /* Juni */
case 9:                          /* September */
case 11: Tage = 30; break;       /* November */
case 2:                          /* Februar */
    if(Schaltjahr) Tage = 29;
    else Tage = 28;
    break;
default: Tage = 31; break; /* sonst */
}
```

Es sei allerdings davor gewarnt zu „trickreich“ zu programmieren. Ein aufgeräumter Quellcode, der für einen anderen Programmierer schnell zu verstehen ist, ist einem ausgefeilten, bis zur Unlesbarkeit optimierten Quellcode zu bevorzugen! Im Zweifelsfall sollten ein paar Kommentarzeilen extra spendiert werden.

4.3 Schleifen

Schleifen jeglicher Art sorgen dafür, dass einige Programmzeilen mehrfach ausgeführt werden. In C stehen uns drei Arten von Schleifen zur Verfügung, die im folgenden erläutert werden: Die *while*- und die *for*-Schleife als kopfgesteuerte Schleife (pre checked loops) sowie die *do*-Schleife als fußgesteuerte Schleife (post checked loop). Zusätzlich gibt es die Befehle *break* und *continue*, welche die Arbeitsweise der Schleifen beeinflussen.

4.3.1 While-Schleife (pre checked loop)

Die *while*-Schleife ist eine kopfgesteuerte Schleife (pre checked loop), das heißt, die Schleifenbedingung wird vor der Ausführung der Schleife geprüft. Das kann also im Sonderfall auch heißen, dass die Schleife nicht ein einziges mal ausgeführt wird.

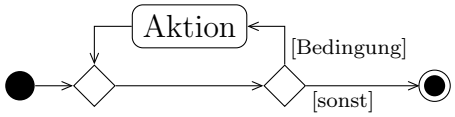


Abbildung 4.7: Aktivitätsdiagramm einer kopfgesteuerten Schleife (*while* und *for*).

Die Syntax für eine while-Schleife sieht wie folgt aus:

while(*Schleifenbedingung*) *Aktion*
Beispiel: Ein Programm soll die Ausgabe von 50,- € Scheinen derart veranlassen, dass das Konto im Plus bleibt:

```
Kontostand = 345.23;
while(Kontostand>=50) {
    printf("Fünfzig Euro ausgeben.\n");
    Kontostand -= 50;
}
```

Hier wird gleich am Anfang gefragt, ob sich noch genug Geld auf dem Konto befindet. Wenn ja, findet die Geldausgabe statt (hier nur eine Textausgabe), wenn

nein, wird die Schleife beendet. Ist der Kontostand gleich am Anfang unter 50,- € oder gar im negativen, so findet keine Ausgabe statt.

Bei allen Schleifen muss darauf geachtet werden, dass es ein sinnvolles Ende der Schleife gibt. Voraussetzung dafür ist, dass mindestens eine Größe der Schleifenbedingung innerhalb der Schleife verändert wird (im Beispiel die Variable *Kontostand*).

4.3.2 For-Schleife (pre checked loop)

Die *for*-Schleife ist der *while*-Schleife sehr ähnlich. In einer typischen *while*-Schleife wird vor Beginn eine Variable initialisiert, es gibt eine Schleifenbedingung und die Variable wird in der Schleife verändert. Bei der *for*-Schleife werden diese drei Schritte in einer Zeile erledigt. Die Syntax der *for*-Schleife lautet:

for(*Initialisierung*; *Schleifenbedingung*; *Änderung*) *Aktion*

Als Beispiel soll die Ausgabe von 50,- € Scheinen mit einer *for*-Schleife realisiert werden:

```
for(Kontostand=345.23; Kontostand>=50; Kontostand-=50)
    printf("Fünfzig Euro ausgeben.\n");
```

Die *for*-Schleife wird bei einfachen Bedingungen bevorzugt. Wenn zum Beispiel in einer Schleife alle Zahlen von eins bis hundert der Reihe nach benötigt werden, ist die *for*-Schleife die richtige Wahl. Bei komplizierten Änderungen der zu prüfenden Variablen ist die *while*-Schleife besser geeignet.

Bei der *for*-Schleife können auch Felder leer gelassen werden. Wenn zum Beispiel der Kontostand schon vor der Schleife gegeben ist, könnte der Code wie folgt aussehen:

```
for(; Kontostand>=50; Kontostand-=50)
    printf("Fünfzig Euro ausgeben.\n");
```

Eine fehlende Schleifenbedingung wird als *wahr* interpretiert. Mehrere Initialisierungen oder Änderungen können durch Kommata getrennt eingegeben werden.

4.3.3 Do-Schleife (post checked loop)

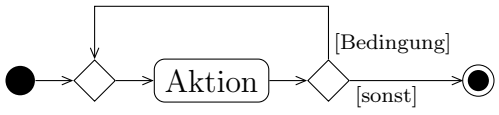


Abbildung 4.8: Aktivitätsdiagramm einer fußgesteuerten Schleife (*do*).

Die *do*-Schleife unterscheidet sich von den vorherigen beiden derart, dass die Schleifenbedingung erst am Ende der Schleife geprüft wird, und dadurch die Schleife mindestens einmal ausgeführt wird, siehe Abbildung 4.8. Die Syntax lautet:

do *Aktion* **while**(*Schleifenbedingung*);

Die *do*-Schleife ist zum Beispiel bei Benutzerabfragen mit anschließender Prüfung der Eingabe sinnvoll:

```
do {
    printf("Geben Sie eine ganze Zahl größer 0 ein:");
    scanf("%d", &Zahl);
} while(Zahl<=0);
```

4.3.4 Unterbrechung von Schleifen (break und continue)

Wir haben zwei Schleifen kennengelernt, bei denen die Schleifenbedingung gleich am Anfang geprüft wird, und eine Schleife, wo diese Prüfung am Ende erfolgt. Ab und zu kann es sinnvoll sein, eine Schleife irgendwo in der Mitte vorzeitig zu beenden. Zu diesem Zweck stehen zwei Befehle zur Verfügung: *break* und *continue*.

Mit dem Befehl *break* kann eine Schleife an beliebiger Stelle, ohne erneutes Prüfen der Schleifenbedingung, beendet werden. Beispiel:

```
/* nicht so schönes Beispiel */
for( ; ; ) {
    printf("Geben Sie eine ganze Zahl größer 0 ein:");
    scanf("%d", &Zahl);
    if(Zahl>0) break;
    printf("Sie haben eine zu kleine Zahl eingegeben!\n");
}
```

Der Befehl *continue* überspringt den Rest der Anweisungen innerhalb der Schleife, so dass eine erneute Überprüfung der Schleifenbedingung erfolgt. Beispiel:

```
/* nicht so schönes Beispiel */
do {
    printf("Geben Sie eine ganze Zahl größer 0 ein (0=Ende):");
    scanf("%d", &Zahl);
    if(Zahl<=0) continue;
    >>> Verarbeitung der Zahl <<<
} while(Zahl!=0);
```

Auch wenn die Befehle *break* und *continue* an einigen Stellen recht elegant sind, sollte man sie um der Lesbarkeit der Programme willen nicht verwenden.

4.3.5 Absolute Sprünge (goto und Marken)

Die absoluten Sprünge sind nur der Vollständigkeit halber immer noch in C vorhanden und sollen auch nur deshalb hier erwähnt werden. Mehr dazu am Ende dieses Abschnitts.

Mit *goto* kann von einer beliebigen Stelle zu einer beliebigen anderen Stelle mit einer Marke innerhalb der gleichen Funktion gesprungen werden. Eine Marke hat die gleiche Form wie ein Variablenname, gefolgt von einem Doppelpunkt. Beispiel:

```
for( ... ) {
    for( ... ) {
        if(schwerer Fehler) goto Fehler;
```

```
        }
    }
    ...
Fehler:
    printf("Es ist ein schwerer Fehler aufgetreten!\n");
```

Grundsätzlich lassen sich alle *goto*-Befehle durch etwas mehr Code umgehen. Der Große Nachteil der *goto*-Befehle liegt darin, dass sie Programme beliebig unübersichtlich werden lassen. Einer Marke kann nicht angesehen werden, von wo aus sie aufgerufen wird. Von daher lässt sich der Programmcode nur schwer rückwärts verfolgen.

4.4 Aufgaben

Aufgabe 4.1: Wie werden in C mehrere Anweisungen zu einem Block zusammengefasst?

Aufgabe 4.2: Implementieren Sie eine einfache Verzweigung, die bei einer negativen Zahl i das Vorzeichen von i umkippt.

Aufgabe 4.3: Implementieren Sie eine einfache Alternative, die bei Werten von i kleiner null den Text "negativ" und sonst den Text "nicht negativ" ausgibt.

Aufgabe 4.4: Erstellen Sie eine mehrfache Alternative, die in Abhängigkeit eines Index i mit Werten eins bis vier die Worte "Gabel", "Messer", "Essloeffel" oder "Teeloeffel" ausgibt.

Aufgabe 4.5: Erklären Sie die Begriffe *fußgesteuerte* Schleife und *kopfgesteuerte* Schleife.

Aufgabe 4.6: Implementieren Sie eine Schleife, die zehnmal das Wort "Hallo" untereinander ausgibt.

Aufgabe 4.7: Erstellen Sie eine Schleife, die für alle ganzen Zahlen von eins bis zwanzig die Quadrate berechnet und ausgibt.

Aufgabe 4.8: Erstellen Sie eine Schleife, in der eine Gleitkommazahl ausgehend von dem Wert eins solange ausgegeben und halbiert wird, bis der Wert null herauskommt.

Aufgabe 4.9: Warum sollten absolute Sprünge vermieden werden?

Kapitel 5

Funktionen

5.1 Funktionen (Unterprogramme)

Der Sprachumfang des Kerns von C ist recht eingeschränkt und umfasst nur wenige Schlüsselwörter. Durch Bibliotheken können aber zahlreiche Funktionen zugeladen werden. Wir haben bereits die Bibliothek *stdio* mit den Funktionen *printf* und *scanf* kennengelernt. Andere Bibliotheken sind z.B. *math* für mathematische Funktionen, *string* zur Verarbeitung von Zeichenketten und *alloc* zur dynamischen Speicherverwaltung.

C bietet aber auch die Möglichkeit eigene Funktionen zu definieren. Von solchen selbstdefinierten Funktionen handelt dieser Abschnitt. Zunächst betrachten wir einfache Funktionen (ohne Parameter und Rückgabewerte), wollen dann einer Funktion Parameter übergeben, werden dann einen einzelnen Wert einer Funktion zurückgeben bevor wir schließlich mehrere Werte aus einer Funktion zurückgeben.

5.1.1 Einfache Funktionen

In größeren Programmen gibt es häufig Programmstücke, die an mehreren Stellen benötigt werden. Anstatt die gleichen Zeilen immer wieder zu implementieren, gibt es die Möglichkeit, kleine Unterprogramme, sogenannte Funktionen zu erstellen.

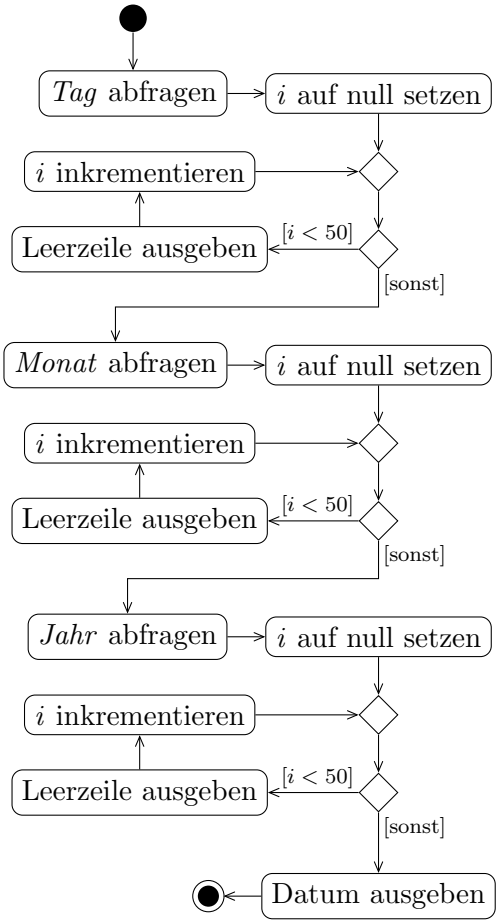
In dem folgenden Programm wird vom Benutzer ein Datum abgefragt. Dabei soll jeweils nach Abfrage von Tag, Monat und Jahr der Bildschirm gelöscht werden, indem fünfzig Leerzeilen ausgegeben werden. Ohne eine Funktion könnte die Aufgabe wie folgt realisiert werden, siehe Abbildung 5.1.

Nach jeder Abfrage wird eine *for*-Schleife durchlaufen, in der fünfzig Zeilenvorschübe ausgegeben werden. Das gleiche kann einfacher mit Hilfe einer Funktion realisiert werden, siehe Abbildung 5.2.

Aus dem Struktogramm wird deutlich, es gibt eine Funktion, die den Bildschirm löscht. Diese Funktion wird an drei Stellen im Hauptprogramm aufgerufen, jeweils nach den Benutzerabfragen. Der zweite Block beschreibt die Funktion, mit der fünfzig Leerzeilen ausgegeben werden.

Im C-Programm in Abbildung 5.2 sehen wir im unteren Teil die Definition der Funktion. Die Definition einer Funktion beginnt immer mit der Beschreibung des Ein- und Ausgabeverhaltens der Funktion, in unserem Beispiel:

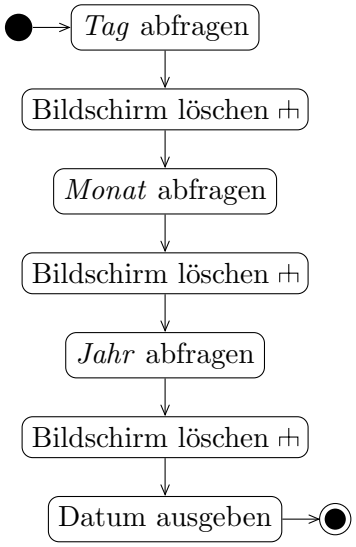
```
void ClrScr(void)
```




```c
#include <stdio.h>
int main()
{
    int Tag;    /* aktueller Tag */
    int Monat;  /* aktueller Monat */
    int Jahr;   /* aktuelles Jahr */
    int i;      /* Laufvariable */

    printf("Tag des Datums: ");
    scanf("%d", &Tag);
    for(i=0; i<50; i++)
        printf("\n");

    printf("Monat des Datums: ");
    scanf("%d", &Monat);
    for(i=0; i<50; i++)
        printf("\n");

    printf("Jahr des Datums: ");
    scanf("%d", &Jahr);
    for(i=0; i<50; i++)
        printf("\n");

    printf("Datum: %d.%d.%d\n",
        Tag, Monat, Jahr);

    return 0;
}
```

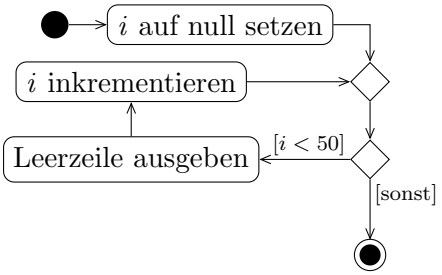
Abbildung 5.1: Beispiel eines Programms ohne Funktion (Unterprogramm).

Hauptprogramm:




```c
#include <stdio.h>
void ClrScr();
int main()
{
    int Tag;    /* aktueller Tag */
    int Monat;  /* aktueller Monat */
    int Jahr;   /* aktuelles Jahr */

    printf("Tag des Datums: ");
    scanf("%d", &Tag);
    ClrScr();
    printf("Monat des Datums: ");
    scanf("%d", &Monat);
    ClrScr();
    printf("Jahr des Datums: ");
    scanf("%d", &Jahr);
    ClrScr();
    printf("Datum: %d.%d.%d\n",
        Tag, Monat, Jahr);

    return 0;
}
```

Bildschirm löschen:




```c
void ClrScr()
{
    int i;      /* Laufvariable */

    for(i=0; i<50; i++)
        printf("\n");

    return;
}
```

Abbildung 5.2: Beispiel aus Abbildung 5.1, diesmal mit der Funktion *ClrScr()*.

Der Funktionsname ist *ClrScr* (*clear screen*, zu deutsch *lösche den Bildschirm*). Das *void* vor dem Funktionsnamen deutet an, dass die Funktion keinen Wert zurückgibt. Das *void* in den runden Klammern hinter dem Funktionsnamen bedeutet, dass die Funktion auch keine Parameter übernimmt.

Nach dieser allgemeinen Beschreibung der Funktion erfolgt in geschweiften Klammern der eigentliche Inhalt der Funktion. In unserem Beispiel ist das die *for*-Schleife, mit der fünfzig Leerzeilen ausgegeben werden. Für die Schleife wird innerhalb der Funktion lokal eine Variable definiert. Am Ende der Funktion steht ein *return*. (Da kein Wert zurückgegeben wird, könnte der Befehl an dieser Stelle auch weggelassen werden.)

Der Aufruf der Funktion *ClrScr* erfolgt durch Angabe des Namens gefolgt von leeren runden Klammern, da keine Parameter übergeben werden. Der Aufruf wird wie alle Anweisungen mit einem Semikolon abgeschlossen.

Wenn der Übersetzer (Compiler) das Programm übersetzt, so braucht er vor dem ersten Aufruf der Funktion Informationen über die Existenz und Beschaffenheit der Funktion. Eine mögliche, aber unübliche Struktur des Programms wäre es, die Definition der Funktion vor das Hauptprogramm *main* zu setzen. Allerdings gibt es Situationen, in denen sich Funktionen gegenseitig (oder sich selbst) aufrufen (rekursive Programmierung), bei denen diese Struktur versagt.

In unserem Beispielprogramm wird eine andere Struktur gewählt: Gleich am Anfang wird die Existenz und Beschaffenheit der Funktion *deklariert*. Das heißt, der Name der Funktion, die zu übergebenden Parameter und der Rückgabewert werden festgelegt. Die *Definition* der Funktion erfolgt später.

In unserem Beispiel sieht die *Deklaration* der Funktion exakt genauso aus wie die *Definition* der Funktion, nur dass die Deklaration durch ein Semikolon abgeschlossen wird.

Nochmal zur Verdeutlichung: Am Anfang steht ein Prototyp der Funktion (*void ClrScr(void);*) den wir *Deklaration* nennen. Das heißt, hier wird *deklariert*, dass es eine Funktion dieser Art geben wird. Die *Definition* der Funktion ist die eigentliche Implementierung der Funktion und folgt später. Das heißt, hier wird *definiert* was die Funktion genau machen soll. Bei der *Deklaration* wird die Funktion nur angekündigt, bei der *Definition* wird die Funktion implementiert.

5.1.2 Argumente und Parameter

In dem letzten Beispiel wurde eine Funktion ohne Parameter und Rückgabewert beschrieben, was in der Praxis eher selten vorkommt. In diesem Abschnitt werden wir einer Funktion Argumente übergeben.

Es soll eine Funktion mit dem Namen *fill* erstellt werden, mit der ein beliebiges Zeichen beliebig oft auf dem Bildschirm ausgegeben werden soll. Mit dieser Funktion können z.B. horizontale Trennstriche in einer Anwendung erzeugt werden. Um z.B. fünfzig Sterne auszugeben soll der Aufruf der Funktion folgendermaßen aussehen:

```
fill('*', 50);
```

Die Definition (Implementierung) von *fill* wird wie folgt realisiert:

```
void fill(char Zeichen, int N)
{
```

```
    int i;

    for(i=0; i<N; i++)
        printf("%c", Zeichen);

    return;
}
```

In der ersten Zeile wird wieder das äußere Verhalten der Funktion beschrieben. Der Name der Funktion ist *fill*. Das *void* vor dem Funktionsnamen heißt, dass kein Wert zurückgegeben wird. In den runden Klammern stehen die Parameter, die von der Funktion *fill* erwartet werden. Der erste Parameter ist vom Typ *char* und wird innerhalb der Funktion mit dem Namen *Zeichen* bezeichnet. Der zweite Parameter ist vom Typ *int* und wird mit *N* bezeichnet.

Die Namen der Parameter, *Zeichen* und *N*, haben außerhalb der Funktion keine Bedeutung. Wichtig ist nur, dass die übergebenen Argumente beim Aufruf der Funktion vom gleichen Typ wie die Parameter der Funktion sind. Z.B. wäre auch folgender Aufruf der Funktion fill möglich:

```
char ch='=';
int  count=23;

fill(ch, count);
```

Hier werden der Funktion *fill* als Argumente keine Konstanten, sondern Variablen übergeben. Die Variable *ch* ist vom Typ *char* und die Variable *count* ist vom Typ *int*, so dass die Typen dieser beiden Argumente mit den Typen der Parameter der Funktion *fill* übereinstimmen.

Die Aufrufargumente und die Funktionsparameter werden vom Übersetzer der Reihe nach verglichen. Die Reihenfolge muss von daher eingehalten werden. Der Aufruf

```
fill(50, '*');      /* falsch! */
```

würde zu einer Fehlermeldung vom Übersetzer führen.

Beim Aufruf einer Funktion wird der Inhalt der Aufrufargumente in die Funktionsparameter kopiert. Innerhalb der Funktion wird nicht mit den Aufrufargumenten selbst, sondern nur mit der Kopie dieser Werte gearbeitet. Die Funktion *fill* könnte z.B. auch wie folgt realisiert werden:

```
void fill(char Zeichen, int N)
{
    while(N--) printf("%c", Zeichen);
    return;
}
```

In der *while*-Schleife wird der Parameter *N* solange um eins verringert, bis er den Wert null annimmt. Beachte, dass der Dekrement-Operator '--' hinter dem Parameter *N* steht, so dass erst geprüft wird, ob er noch ungleich null ist, bevor er dann reduziert wird.

Wird nun die Funktion mit *fill(ch, count);* aufgerufen, so bleibt die Variable *count* unverändert, da die interne Variable *N* nur eine Kopie des aufrufenden Arguments *count* ist.

5.1.3 Rückgabewert einer Funktion

Eine Funktion kann auch einen Rückgabewert haben. Ein Beispiel ist die Sinusfunktion *sin(x)*. Sie erwartet als Argument einen Winkel in Bogenmaß (2π Bogenmaß = 360°), und liefert als Rückgabewert den Sinus des Winkels. Die Deklaration sieht wie folgt aus:

```
double sin(double x);
```

Der Name der Funktion ist *sin*. In den runden Klammern finden wir die Parameter der Funktion, hier der Parameter x vom Typ *double*. Vor dem Funktionsnamen finden wir die Typenbezeichnung *double*, die angibt, dass eine Zahl vom Typ *double* zurückgegeben wird.

Die Rückgabe des Wertes erfolgt mit dem Befehl *return*. Eine Mini-Funktion zur Ermittlung der Zahl π könnte wie fogt aussehen:

```
double pi(void)
{
    return 3.141;
}
```

Soll nun zum Beispiel die Fläche des Kreises A mit bekanntem Radius r berechnet werden ($A = \pi r^2$), so kann dies mit Hilfe der erstellten Funktion erfolgen:

```
A = pi()*r*r;
```

Das folgende Struktogramm zeigt ein Programmstück, indem vom Benutzer das aktuelle Datum abgefragt werden soll. Für die Angaben Jahr, Monat und Tag sollen jeweils Plausibilitätsprüfungen gemacht werden, und die Abfrage ggf. wiederholt werden.

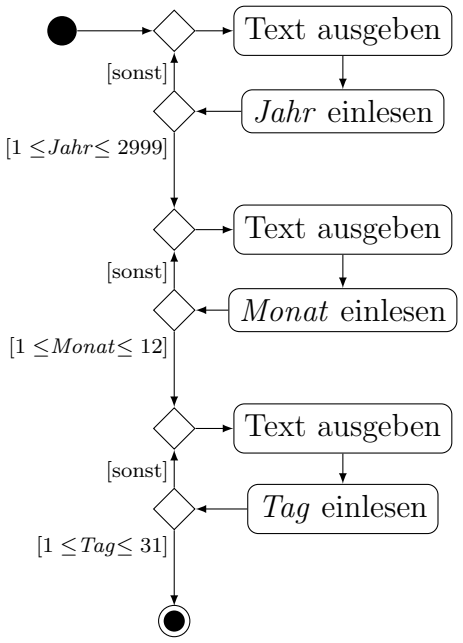


Abbildung 5.3: Programmstück zur Abfrage eines Datums.

Die Abfrage der drei Zahlen sieht jeweils recht ähnlich aus: Es wird ein Text ausgegeben und eine Zahl abgefragt, bis die Zahl innerhalb des gewünschten Bereichs liegt. Es soll nun eine Funktion erstellt werden, die diese Aufgabe übernimmt. Die Funktion braucht als Parameter den Fragetext und die Unter- und Obergrenze der einzugebenden Zahl. Zurückgegeben wird das Ergebnis der Abfrage. Die Definition sieht wie folgt aus:

```
int GetInt(char Text[], int Min, int Max)
{
    int Zahl;

    do {
        printf("%s (%d-%d): ", Text, Min, Max);
        scanf("%d", &Zahl);
    } while(Zahl<Min || Zahl>Max);

    return Zahl;
}
```

Im Kopf der Funktion finden wir den Namen der Funktion *GetInt*, den Rückgabetyp *int* und die Liste der Parameter *char Text[], int Min, int Max*. Der Parameter *char Text[]* deutet eine Zeichenkette unbekannter Länge an. Das oben genannte Programmstück sieht mit der Funktion *GetInt* folgendermaßen aus:

```
Jahr = GetInt("Geben Sie das Jahr ein", 1, 2999);
Monat = GetInt("Geben Sie den Monat ein", 1, 12);
Tag = GetInt("Geben Sie den Tag ein", 1, 31);
printf("Datum: %d.%d.%d\n", Tag, Monat, Jahr);
```

5.1.4 Rückgabe mehrerer Werte

Vom normalen Aufbau her kann eine Funktion nur einen Wert zurückgeben. In manchen Situationen sollen aber mehr als ein Wert als Ergebnis zurückgeliefert werden. Die Parameter einer Funktion sind immer nur Kopien der Aufrufargumente (siehe Abschnitt 5.1.2), so dass die Veränderung der Parameter die aufrufenden Argumente nicht beeinflussen kann.

Das Problem wird gelöst, indem ein Zeiger auf die zu ändernde Variable übergeben wird. Auch wenn von diesem Zeiger wieder eine Kopie erstellt wird, so zeigt dieser Zeiger doch auf die selbe Stelle im Speicher, wie das Argument beim Aufruf der Funktion.

Auch wenn das Thema Zeiger erst zu einem späteren Zeitpunkt behandelt wird, hier ein Beispiel. Die Funktion *swapi* soll zwei Zahlen vom Typ *int* vertauschen. Die Definition wird wie folgt realisiert:

```
void swapi(int *a, int *b)
{
    int x;

    x = *a;
    *a = *b;
    *b = x;
}
```

Vor dem Funktionsnamen *swapi* steht der Typ *void*, der besagt, dass kein Wert direkt zurückgegeben wird. Bei den Parametern *int *a* und *int *b* bedeuten die Sterne vor den Parameternamen, dass ein *Zeiger auf den Typ int* übergeben wird. Innerhalb der Funktion kann auf den Inhalt des Zeigers, also auf den Wert, auf den der Zeiger zeigt, mit einem Stern vor der Variable zugegriffen werden. Ein Aufruf würde so aussehen:

```
int Zahl1=5;
int Zahl2=7;

swapi(&Zahl1, &Zahl2);
```

Das kaufmännische Und vor einer Variable liefert die Speicheradresse der Variable, die hier an die Funktion *swapi* übergeben wird. Nach dem Aufruf hat die Variable *Zahl1* den Wert 7, und die Variable *Zahl2* den Wert 5.

5.2 Globale und lokale Variablen

In C wird zwischen *globalen* und *lokalen* Variablen unterschieden. Globale Variablen sind überall bekannt und verfügbar; sie sind *global*. Lokale Variablen sind nur in einem beschränkten Bereich bekannt und verfügbar; sie sind *lokal*. Eine andere Bezeichnung für global und lokal ist *extern* und *intern*.

5.2.1 Lokale Variablen

Bisher haben wir Variablen nur innerhalb von Funktionen definiert. (Das Hauptprogramm *main* wird auch als Funktion behandelt.) Eine solche Variable ist nur innerhalb der jeweiligen Funktion bekannt. Wenn Sie außerhalb einer Funktion auf eine ihrer lokalen Variablen versuchen zuzugreifen, so wird der Übersetzer einen Fehler melden.

Da lokale Variablen nur in der eigenen Funktion bekannt sind, können Namen in unterschiedlichen Funktionen auch doppelt verwendet werden. Der Übersetzer wird sie als unterschiedliche Variablen behandeln.

Der Vorteil von lokalen Variablen ist: Wenn sie eine Funktion fertiggestellt haben, brauchen Sie sich um die lokalen Namen keine Gedanken mehr machen. Sie können woanders wieder verwendet werden, ohne dass es zu Problemen kommt.

5.2.2 Globale Variablen

Globale Variablen werden außerhalb von Funktionen definiert. Wenn Sie z.B. direkt vor dem Hauptprogramm *main* die Variable *pi* wie folgt definieren,

```
double pi=3.141;
```

so kann in allen Funktionen des Programms von dieser Variable Gebrauch gemacht werden.

Wird in einer Funktion eine lokale Variable mit demselben Namen einer globalen Variable definiert, so erstellt der Übersetzer zwei getrennte Variablen. Innerhalb der

besagten Funktion steht der Name dann für die lokale Variable, von allen anderen Teilen aus steht der Name für die globale Variable.

Das folgende Beispielprogramm fasst die wesentlichen Eigenschaften von lokalen und globalen Variablen zusammen:

```
#include <stdio.h>

void Funktion1();
void Funktion2();

int Zahl;

int main()
{
    Zahl = 25;

    printf("Main:␣␣␣␣␣␣Zahl=%d\n", Zahl);
    Funktion1();
    Funktion2();
    Funktion1();
    printf("Main:␣␣␣␣␣␣Zahl=%d\n", Zahl);

    return 0;
}

void Funktion1()
{
    printf("Funktion1:␣Zahl=%d\n", Zahl);
}

void Funktion2()
{
    int Zahl;

    Zahl = 17;
    printf("Funktion2:␣Zahl=%d\n", Zahl);
}
```

Wenn Sie dieses Programm starten, erhalten Sie folgende Ausgabe auf dem Bildschirm:

```
Main:      Zahl=25
Funktion1: Zahl=25
Funktion2: Zahl=17
Funktion1: Zahl=25
Main:      Zahl=25
```

5.2.3 Verwendung von lokalen und globalen Variablen

Grundsätzlich gilt: meiden Sie globale Variablen. Das heißt, wenn Sie die Wahl zwischen globalen und lokalen Variablen haben, so ist die lokale Variable immer zu bevorzugen. Warum? Eine globale Variable kann von allen Programmteilen gelesen und verändert werden. Das lässt sich im Nachhinein nur sehr schwer nachvollziehen. Programme mit globalen Variablen sind schwerer zu verstehen und für Fehler anfälliger.

5.3 Aufgaben

Aufgabe 5.1: Implementieren Sie eine Funktion mit Namen *hello*, die den Text "hello world" mit einem anschließenden Zeilenvorschub auf den Bildschirm ausgibt.

Aufgabe 5.2: Deklarieren Sie eine Funktion mit Namen *Betrag*, den Parametern x und y vom Typ *double* und einem Rückgabewert vom Typ *double*.

Aufgabe 5.3: Wie kann eine Funktion mehrere Variablen der aufrufenden Funktion modifizieren?

Aufgabe 5.4: Erstellen Sie eine Funktion, die zwei Gleitkommazahlen vom Typ *double* vertauscht.

Aufgabe 5.5: Erläutern sie die Begriffe *globale* und *lokale Variablen*.

Aufgabe 5.6: Warum sollten globale Variablen vermieden werden?

Kapitel 6

Vektoren (Arrays)

In C können Datentypen verkettet werden. Eine solche Kette eines Typs wird als *Vektor* bezeichnet. Es gibt viele Beispiele für Vektoren: Angefangen bei einem Text, hinter dem sich ein Vektor von Zeichen verbirgt (eindimensional), über die Darstellung einer mathematischen Funktion durch eine Liste (Vektor) von Gleitkommazahlen (eindimensional) bis hin zu mehrdimensionalen Gebilden wie die Pixel eines Bildes (zweidimensional), die Temperaturverteilung in einem beheizten Raum (dreidimensional) oder der Zeitliche Verlauf der Temperatur in einem Raum wenn das Fenster geöffnet wird (vierdimensional). In den folgenden zwei Abschnitten wird das Arbeiten mit Vektoren erläutert.

Eine Variable wird häufig mit vier Eigenschaften beschreiben:

Typ. Eine Variable hat in C einen definierten Datentyp wie z.B. **int** oder **char**.

Name. Um Variablen von anderen zu unterscheiden wird ihnen ein eindeutiger Name zugewiesen.

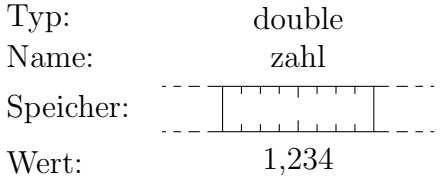
Adresse/Speicher. Eine Variable wird an einer Stelle im Speicher abgelegt, die durch eine eindeutige Adresse definiert ist.

Wert. Schließlich hat die Variable auch einen Wert.

Wir betrachten z.B. folgende Definition:

```
double zahl=1.234;
```

Die vier Eigenschaften der Variable zahl können so dargestellt werden:



Um das Thema Vektoren und später auch Zeiger (pointer) besser zu verstehen, werden wir uns in diesem Kapitel immer wieder die vier Eigenschaften der Variablen vergegenwärtigen.

6.1 Eindimensionale Vektoren

6.1.1 Definition eines eindimensionalen Vektors

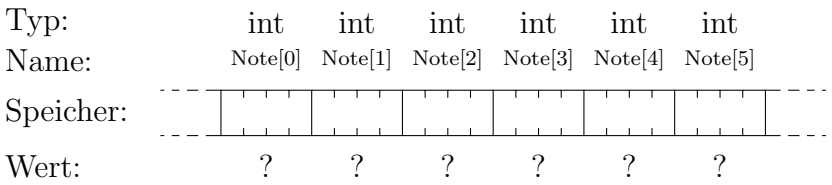
Die allgemeine Syntax zur Definition eines eindimensionalen Vektors lautet:

```
<Datentyp> <Variablenname>[<Anzahl>];
```

Beispiel: Sie wollen zur Erstellung eines Notenspiegels die Anzahl der Einsen, der Zweien ... und der Sechsen ermitteln. Dafür benötigen Sie einen Vektor mit sechs ganzen Zahlen:

```
int Note[6];
```

Ab dieser Zeile wird im Arbeitsspeicher Platz für sechs **int**-Variablen geschaffen. Die folgende Skizze fasst die vier Eigenschaften für die sechs Variablen zusammnen. Der Wert der Variablen ist noch unbekannt.



6.1.2 Initialisierung eines eindimensionalen Vektors

Um einen eindimensionalen Vektor zu initialisieren gibt es zwei Möglichkeiten. Erstens, Sie weisen jedem Element des Vektors nach der Definition einzeln einen Wert zu. Am Beispiel der Noten kann das entweder so,

```
Note[0] = 0;
Note[1] = 0;
Note[2] = 0;
Note[3] = 0;
Note[4] = 0;
Note[5] = 0;
```

oder, und das ist vor allem für große Vektoren interessant, besser so aussehen:

```
for(i=0; i<6; i++) Note[i] = 0;
```

Hier kann bereits gesehen werden, wie die Elemente eines Vektors angesprochen werden. Nach dem Namen wird in eckigen Klammern die Nummer (der Index) des Elements angegeben. Beachte: Die Zählung beginnt bei *null* und hört eins *vor* der Anzahl der Elemente auf! Unser Vektor für die Noten hat sechs Elemente mit den Indizes *null* bis *fünf*.

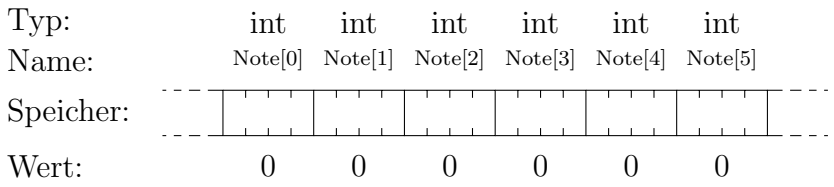
Die zweite Möglichkeit der Initialisierung ist direkt bei der Definition. In unserem Beispiel sieht das wie folgt aus:

```
int Note[6] = { 0, 0, 0, 0, 0, 0 };
```

In dieser Schreibweise steckt eine Redundanz: Die Anzahl der Elemente wird sowie durch die Zahl in den eckigen Klammern, als auch durch die Anzahl der Elemente in den geschweiften Klammern angegeben. Es ist von daher nur logisch, dass die Zahl in den eckigen Klammern weggelassen werden kann:

```
int Note[] = { 0, 0, 0, 0, 0, 0 };
```

Nach der Initialisierung sind die vier Eigenschaften der sechs Variablen definiert. Es ergibt sich folgende Skizze:



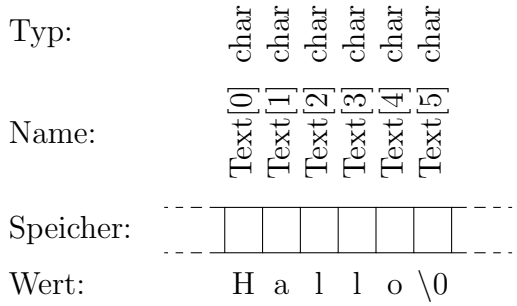
Bei Zeichenketten gibt es eine Sonderregel. Die Initialisierung

```
char Text[] = { 'H', 'a', 'l', 'l', 'o', '\0' };
```

kann verkürzt werden zu

```
char Text[] = "Hallo";
```

und es ergeben sich die folgenden vier Eigenschaften:



6.1.3 Arbeiten mit eindimensionalen Vektoren

Wie wir bereits bei der Initialisierung gesehen haben, werden die Elemente eines Vektors durch eine Zahl in eckigen Klammern hinter dem Vektornamen angesprochen. Das gilt gleichermaßen für Schreib- und Lesezugriffe. Beispiel:

```
Note[2]++;
```

erhöht die Anzahl der Dreien um eins.

```
Bestanden = Note[0]+Note[1]+Note[2]+Note[3];
```

ermittelt die Anzahl aller bestandenen Studenten.

Auch der Index kann Ergebnis eines Ausdrucks sein. Beispiel:

```
if(Zahl>=1 && Zahl<=6) Note[Zahl-1]++;
```

C überprüft nicht die Grenzen des Vektors. Beim Beispiel der Noten führt

```
Zahl = Note[6];    /* Fehler */
```

zu einem unsinnigen Ergebnis. Schlimmer noch,

```
Note[6] = Zahl;    /* schlimmer Fehler */
```

kann zum Absturz Ihres Programms oder bei unsoliden Betriebssystemen zum Absturz des Computers führen. Der Grund liegt in der besonderen Stärke von C: der Geschwindigkeit. Würden bei jedem Zugriff auf einen Vektor zuerst die Grenzen geprüft, so würde das viel Rechenzeit kosten. In C ist es daher Aufgabe des Programmierers, die Grenzen von Vektoren zu wahren.

6.2 Mehrdimensionale Vektoren

Mehrdimensionale Vektoren sind die logische Erweiterung der eindimensionalen Vektoren. Ein zweidimensionaler Vektor ist eine Verkettung von eindimensionalen Vektoren. Ein dreidimensionaler Vektor ist eine Verkettung von zweidimensionalen Vektoren u.s.w.

6.2.1 Definition eines mehrdimensionalen Vektors

Die Anzahl der Dimensionen wird durch die Anzahl der Zahlen in eckigen Klammern bestimmt. Daher lautet die allgemeine Definition eines zweidimensionalen Vektors:

```
<Datentyp> <Variablenname>[<Anzahl>][<Anzahl>];
```

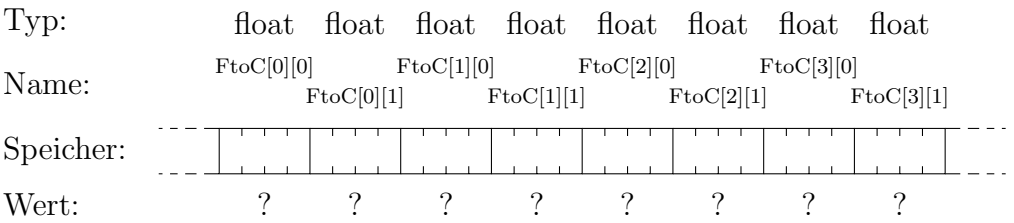
Die allgemeine Definition eines dreidimensionalen Vektors lautet:

```
<Datentyp> <Variablenname>[<Anzahl>][<Anzahl>][<Anzahl>];
```

u.s.w. Beispiel: Es soll eine Umrechnungstabelle für Temperaturen von Fahrenheit nach Celsius mit vier Zeilen erstellt werden.

```
float FtoC[4][2];
```

Die acht Elemente der Tabelle sind wie folgt im Speicher abgelegt. Die Werte sind uns noch unbekannt.



Der Arbeitsspeicher eines Computers ist eindimensional organisiert. Man kann sich den Speicher wie eine sehr lange Papierrolle vorstellen, wie sie in den Kassen im Supermarkt verwendet werden. Es passt immer nur eine Zahl in eine Zeile, so dass alle Zahlen sequentiell untereinander geschrieben werden müssen. Auch wenn die Zahlen, wie in unserem Beispiel, ein zweidimensionales Feld beschreiben, werden sie hintereinander im Speicher abgelegt.

6.2.2 Initialisierung eines mehrdimensionalen Vektors

Auch hier kann die Initialisierung bei der Definition erfolgen. Da der zweidimensionale Vektor eine Verkettung eindimensionaler Vektoren ist, wird auch die Initialisierung entsprechend verkettet. Am Beispiel der Umrechnungstabelle sieht das so aus (Umrechnung: $\frac{\vartheta_C}{^\circ C} = \frac{5}{9}(\frac{\vartheta_F}{^\circ F} - 32)$):

```
float FtoC[4][2] = { {   0, -17.78f },
                     {  50,   10    },
                     { 100,  37.78f },
                     { 150,  65.56f } };
```

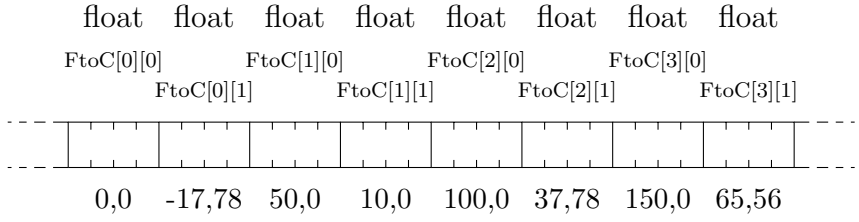
(Das f hinter den Zahlen mit Nachkommastelle verhindert, dass der Compiler Warnungen ausgibt. Ohne das f würden die Zahlen als **double** mit ca. 15 tragenden Stellen interpretiert, die mit Verlusten in den Typ **float** umgewandelt werden.) Die Elemente eines mehrdimensionalen Vektors können auch einzeln initialisiert werden:

```
FtoC[0][0] = 0;
FtoC[0][1] = -17.78f;
FtoC[1][0] = 50;
    u.s.w.
```

Auch die Verwendung einer Schleife kann nützlich sein:

```
for(i=0; i<4; i++) {
    FtoC[i][0] = (float)(50*i);
    FtoC[i][1] = (float)(5.0/9.0*(FtoC[i][0]-32));
}
```

(Um Warnungen vom Compiler zu vermeiden werden die Ergebnisse explizit in **float** umgewandelt.) Nach der Initialisierung sieht der zweidimensionale Vektor mit seinen vier Eigenschaft wie folgt aus:



6.2.3 Arbeiten mit mehrdimensionalen Vektoren

Der Zugriff auf die Elemente eines mehrdimensionalen Vektors erfolgt durch Zahlen in eckigen Klammern hinter dem Vektornamen. Die Anzahl der Zahlen muss mit der Anzahl der Dimensionen des Vektors übereinstimmen.

Beim Beispiel der Umrechnung von Fahrenheit in Celsius kann das so aussehen:

```
for(i=0; i<4; i++)
    printf("%6.2f F -> %6.2f C\n", FtoC[i][0], FtoC[i][1]);
```

Es ergibt sich folgende Ausgabe:

```
  0.00 F -> -17.78 C
 50.00 F ->  10.00 C
100.00 F ->  37.78 C
150.00 F ->  65.56 C
```

Auch hier gilt, die Grenzen des Vektors müssen vom Programmierer überwacht werden. Bei Missachtung kann das verheerende Folgen haben.

6.3 Vektoren als Funktionsparameter

Vektoren können wie andere Variablen als Parameter an Funktionen übergeben werden. Ein wichtiger Unterschied ist: Ein Vektor wird *nicht* als Kopie übergeben. Verändert die Funktion den Vektor-Parameter, so wird auch die übergebene Variable der aufrufenden Funktion modifiziert.

6.3.1 Eindimensionale Vektoren als Parameter

Ein Vektor kann als Parameter einer Funktion übergeben werden. Beispiel:

```
void Init(int Note[6]);

int main()
{
    int Note_Sem_1[6];

    Init(Note_Sem_1);

    return 0;
}

void Init(int Note[6])
{
    int i;

    for(i=0; i<6; i++) Note[i] = 0;
}
```

So wie wir es für andere Datentypen kennengelernt haben, wird die Variable in runden Klammern hinter den Funktionsnamen eingetragen. Der Name innerhalb der Funktion (hier *Note*) darf sich von dem Namen der übergebenen Variable (hier *Note_Sem_1*) unterscheiden. Da keine Kopie übergeben wird, sondern das "Original", kann die Funktion direkt die Variable der aufrufenden Funktion modifizieren. (In unserem Beispiel werden alle Vektorelemente auf null gesetzt.)

Die Information, wie viele Elemente der Vektor hat, braucht die Funktion eigentlich nicht. Von daher ist es nur logisch, dass die Größe des Vektors bei der Deklaration und im Funktionskopf weggelassen werden kann:

```
void Init(int Note[]);
```

Das ist z.B. sinnvoll, wenn ein Text verarbeitet werden soll:

```
void Ausgeben(char Text[]);
```

Eine solche Funktion kann Texte beliebiger Länge verarbeiten. (Die Länge ergibt sich durch die abschließende null im Vektor.)

6.3.2 Mehrdimensionale Vektoren als Parameter

Wir wollen die Tabelle aus Abschnitt 6.2 (hier vom Typ *double*) in einer Funktion initialisieren:

```
void InitTemperatur(double FtoC[4][2]);

int main()
{
    double FtoC[4][2];

    InitTemperatur(FtoC);

    return 0;
}

void InitTemperatur(double FtoC[4][2])
{
    int i;

    for(i=0; i<4; i++) {
        FtoC[i][0] = 50.0*i;
        FtoC[i][1] = 5.0/9.0*(FtoC[i][0]-32.0);
    }
}
```

Wir gehen nach dem gleichen Prinzip wie bei eindimensionalen Vektoren vor: Der Vektor wird in die Parameterliste der Funktion eingetragen. Da keine Kopie übergeben wird, kann die Funktion direkt den Vektor der aufrufenden Funktion verändern.

Können wir auch hier die Angabe der Vektorgröße weglassen? Es kann nur die *erste* Angabe zur Dimension leer gelassen werden (hier die Angabe 4). Die anderen Dimensionen (hier die Angabe 2) benötigt der Compiler, um die Position der Vektorelemente im Speicher zu bestimmen. Zur Veranschaulichung siehe die Skizze im Abschnitt 6.2.1. In unserem Beispiel können wir die Deklaration bzw. den Funktionskopf also verkürzen zu:

```
void InitTemperatur(double FtoC[][2]);
```

6.4 Aufgaben

Aufgabe 6.1: Deklarieren Sie einen Vektor mit Namen *Liste* mit zehn Elementen von Typ *double*.

Aufgabe 6.2: Initialisieren Sie alle Elemente des soeben erstellten Vektors mittels einer Schleife mit null.

Aufgabe 6.3: Setzen Sie das letzte Element des oben erstellten Vektors auf 3.

Aufgabe 6.4: Wie viele Bytes benötigt der eben erstellte Vektor im Speicher?

Aufgabe 6.5: Definieren Sie einen Vektor mit den Zahlen 1, 2, 3, 5, 7 und 11.

Aufgabe 6.6: Sie wollen über einen Zeitraum von drei Stunden alle zehn Sekunden die Temperatur in einem Wassertopf erfassen. Erstellen Sie einen geeigneten Vektor zum Speichern und Verarbeiten der Daten.

Aufgabe 6.7: Sie übergeben einen Vektor als Parameter an eine Funktion. Worin unterscheidet sich die Übergabe des Vektors von der Übergabe eines elementaren Datentyps wie *int* oder *double*?

Aufgabe 6.8: Definieren Sie einen zweidimensionalen Vektor für ein Schachbrett mit acht mal acht Feldern. Jedes Element soll vom Typ *int* sein.

Aufgabe 6.9: Initialisieren Sie den eben definierten Vektor mit null.

Aufgabe 6.10: Setzen Sie die Felder in den Ecken auf eins.

Aufgabe 6.11: Wie viele Bytes benötigt der eben erstellte Vektor im Speicher?

Aufgabe 6.12: Wiederholen Sie die letzten vier Aufgaben für einen dreidimensionalen Vektor vom Typ *float* mit 100*100*100 Elementen.

Aufgabe 6.13: Definieren Sie einen Vektor für eine Umrechnungstabelle mit zwanzig Zeilen.

Aufgabe 6.14: Definieren Sie einen Vektor für die Temperaturverteilung in einem Raum der Größe 2x3x4 m. Jeder Kubikdezimeter soll mit einem Wert belegt werden können.

Aufgabe 6.15: Einer Funktion soll ein eindimensionaler Vektor unbekannter Größe übergeben werden. Wie sieht die Deklaration aus? (Keine Zeiger verwenden!)

Aufgabe 6.16: Einer Funktion soll ein zweidimensionaler Vektor mit Zeilenlänge 5 übergeben werden. Wie sieht die Deklaration der Funktion aus? (Keine Zeiger verwenden!)

Kapitel 7

Projekte organisieren

In diesem Abschnitt soll aufgezeigt werden, wie mehrere Programmierer gemeinsam an einer Software arbeiten können. Dafür schauen wir uns zunächst die Schritte beim Übersetzen eines Quellcodes in ein ausführbaren Programm an. Danach untersuchen wir, wie ein Softwareprojekt auf mehrere Dateien verteilt werden kann. Schließlich betrachten wir etwas allgemeiner die Zyklen über die Lebensdauer einer Software.

7.1 Editor, Übersetzer, Linker

Wird ein Programm in einer Übersetzerprogrammiersprache wie C geschrieben, so geschieht das in den folgenden drei Stufen:

1. Eingabe, bzw. Verändern eines Programms mit einem Editor
   Ergebnis: Quellcode (engl. source code)

2. Übersetzen des Quellcodes mit einem Übersetzer
   Ergebnis: Programmstücke in Maschinensprache

3. Verbinden der Programmstücke
   Ergebnis: Ausführbares, von der CPU lesbares Programm

Tritt bei Stufe zwei oder drei ein Fehler auf, so wird wieder bei Stufe eins begonnen. Auch wenn Stufe zwei und drei ohne Fehler überstanden wurden, und die Ausführung läuft nicht zufriedenstellend, wird wieder bei Stufe eins begonnen.

Ein weiteres Werkzeug bei der Fehlersuche während der Ausführung ist der Debugger. (Ein *Bug* ist ein Fehler, ein *Debugger* ist somit ein „Entfehlerer“.) Mit ihm kann das Programm schrittweise ausgeführt, und dabei die Aktionen des Programms genau beobachtet werden.

7.2 Schritte bei Übersetzung eines Programms

Wurde der Quellcode für ein Programm erstellt, so wird danach das Programm in eine ausführbare Datei übersetzt. In einer modernen Programmierumgebung muss dafür üblicherweise nur ein Knopf betätigt werden, aber hinter diesem Knopf verbergen sich mehrere Schritte:

1. Aktionen vor dem Übersetzen durch den *Pre-Compiler*

2. Übersetzen des Quellcodes durch den *Compiler*
3. Zusammenfügen aller Funktionen durch den *Linker*

Abbildung 7.1 zeigt den Ablauf dieser drei Schritte, die in den folgenden drei Unterabschnitten näher erläutert werden. Die drei Schritte können deutlich mehr als das hier Beschriebene beinhalten; es soll hier nur soviel beschrieben werden, dass Sie die nachfolgenden Abschnitte besser verstehen.

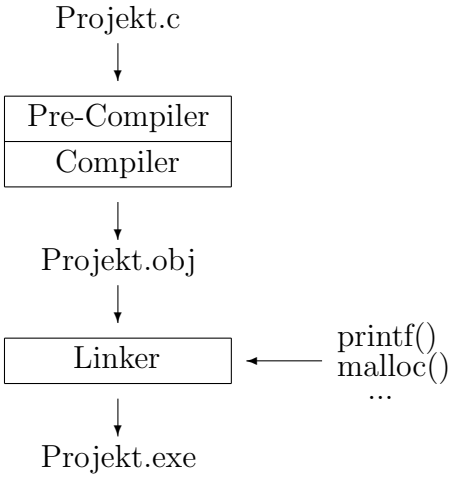


Abbildung 7.1: Übersetzung einer Quellcodedatei

7.2.1 Aktionen vor dem Übersetzen, der *Pre-Compiler*

Der *Pre-Compiler* (Vor-Übersetzer) erledigt einige Aufgaben bevor die eigentliche Übersetzung beginnt. Im Wesentlichen führt der Pre-Compiler alle Pre-Compiler-Befehle aus; das sind in C alle Befehle die mit einem Doppelkreuz '#' beginnen. Der Quellcode wird bearbeitet und für den Compiler vorbereitet.

Der Befehl **#include** <Include−Datei> wird durch den Pre-Compiler entfernt und durch den Inhalt der angegebenen Datei ersetzt. Alle C-Anweisungen zwischen den Befehlen **#if** <Bedingung> und **#endif** werden gelöscht, wenn die Bedingung *nicht wahr* ergibt. Mit dem Befehl **#define** ... wird der Pre-Compiler angewiesen, ab dieser Stelle im Quellcode bstimmte Zeichenfolgen zu ersetzen. — Es gibt noch weitere Befehle für den Pre-Compiler, aber es ist nicht Gegenstand dieses Abschnitts weiter darauf einzugehen.

Erst wenn der Pre-Compiler seine Arbeit abgeschlossen hat, wird mit der Übersetzung des Quelltextes begonnen.

7.2.2 Übersetzen des Quellcodes durch den *Compiler*

Der *Compiler* übersetzt den vom Pre-Compiler vorbereiteten Quelltext, in eine für den Prozessor verständliche Sprache. Das Resultat ist eine Objekt-Datei mit der Endung `obj`.

In dieser Datei befindet sich nur die Übersetzung des erstellten Quellcodes. Wurde eine Funktion (z.B. printf()) aus einer Standard-Bibliotheken (z.B. stdio.h) verwendet, so ist diese noch nicht eingefügt worden. An den entsprechenden Stellen ist nur der Aufruf vermerkt, aber die Funktion selbst fehlt noch.

Der letzte Schritt wird durch den Linker ausgeführt.

7.2.3 Zusammenfügen aller Funktionen durch den *Linker*

Der *Linker* schließlich geht die Objekt-Datei durch, die vom Compiler erstellt wurde, und sucht alle verwendeten externen Funktionen zusammen. Das Ergebnis des Linkers ist eine ausführbare Datei, typischerweise mit der Endung `exe`.

Damit der Linker seine Arbeit erledigen kann, muss er wissen, wo sich die Funktionen befinden, die noch eingebunden werden müssen. Dafür gibt es die *Libraries* (zu deutsch Bibliotheken), das sind Dateien mit der Endung `lib`.

7.3 Preprozessor

Zu einem Übersetzer der Programmiersprache C gehört ein *Preprozessor*, der einige Aufgaben vor dem eigentlichen Übersetzen des Programms übernimmt. Der Preprozessor läuft über den erstellten Quellcode, und nimmt einige Veränderungen vor: Andere Dateien werden eingefügt (*#include*), Makronamen werden durch andere Textstücke ersetzt (*#define*), ganze Programmstücke werden ggf. entfernt (*#if*) etc.

Die Befehle des Preprozessors beginnen alle mit einem Doppelkreuz '#'. Im Folgenden werden die wichtigsten dieser Befehle erläutert.

7.3.1 Dateien einbinden (#include)

Der Preprozessorbefehl *#include* bindet eine Textdatei in den Quellcode ein. Das heißt, die Zeile im Quellcode wird durch den Inhalt der angegebenen Datei ersetzt. Es gibt zwei Versionen:

#include <*Dateiname*>
#include "*Dateiname*"

Wird der Dateiname in spitzen Klammern angegeben, so wird die Datei nur in den eingestellten Pfaden für die Standard-Headerdateien wie *stdio.h* oder *math.h* gesucht. Soll die Suche nach der Datei zunächst in dem Verzeichnis beginnen, in dem sich auch der Quellcode befindet, so muss der Name in Doppelanführungszeichen gesetzt werden. Diese Option wird für selbstgeschriebene Headerdateien verwendet.

7.3.2 Makrodefinition (#define)

Mit *#define* können Sie im Quellcode Makros definieren, die noch vor der eigentlichen Übersetzung vom Preprozessor durch den gewünschten Inhalt ersetzt werden. Es gibt zwei Versionen:

#define *Makroname Makroinhalt*
#define *Makroname* (*Argumentenliste*) *Makroinhalt*

Die erste Version kann z.B. in das Programm aus Abbildung D.11 auf Seite 103 eingebaut werden:

```
#define MAX_ZAHL 100
...
printf("Primzahlen zwischen 1 und %d:\n", MAX_ZAHL);
...
```

```
Prob = 2;
while(Prob<=MAX_ZAHL) {
...
```

Nach der Zeile mit *#define* wird der Preprozessor alle Stellen, an denen er die Bezeichnung *MAX_ZAHL* findet, die Bezeichnung durch die Zeichenfolge *100* ersetzen.

Es ist üblich Makronamen in Großbuchstaben zu schreiben, damit man sie leichter von sonstigen Namen unterscheiden kann.

Bei der zweiten Version werden dem Makro Argumente übergeben. Beispiel:

```
#define MAX(A, B) ((A)>(B)?(A):(B))
```

Der folgende Aufruf führt zu dem gewünschten Resultat von 23:

```
a = 20;
b = 3;
c = MAX(a+b, a-b);
```

Hier wird die Zeile vom Preprozessor wie folgt ersetzt:

```
c = ((a+b)>(a-b)?(a+b):(a-b));
```

An diesem Beispiel wird eine Grenze von Makros deutlich. In der entstandenen Zeile wird der Ausdruck *a+b* zweimal ausgewertet: Einmal beim Vergleich und einmal bei der Rückgabe. Bei Konstanten werden die Zahlen gleich vom Übersetzer zusammengefasst. Handelt es sich aber wie in unserem Beispiel um Variablen, so wird die Operation doppelt ausgeführt, was in zeitkritischen Anwendungen unnötig Zeit kostet.

Makros sind nur innerhalb der aktuellen Quelldatei gültig. Soll ein Makro nur für einen begrenzten Bereich im Quellcode gültig sein, so kann es mit folgendem Befehl wieder aufgehoben werden:

#undef *Makroname*

7.3.3 Bedingte Übersetzung

Der Preprozessor bietet die Möglichkeit, Teile eines Quellcodes nur bedingt zu übersetzen. Das heißt, die Teile, die nicht übersetzt werden, sind in dem übersetzten Programm nicht existent. Es gibt drei Möglichkeiten, eine Bedingte Übersetzung einzuleiten:

#if *konstante Bedingung*
#ifdef *Bezeichner* (oder **#if** defined *Bezeichner*)
#ifndef *Bezeichner* (oder **#if** !defined *Bezeichner*)

Die erste Variante ähnelt der bedingten Anweisung, aber der Befehl wird mit dem Doppelkreuz # eingeleitet. Die Bedingung folgt ohne Klammern und die Bedingung darf nur aus Konstanten bestehen. (Makronamen sind Konstanten.) Beispiel:

```
#if MODUS==3
```

Die zweite und dritte Variante prüft, ob ein Makro existiert oder nicht. Beispiel: Während der Erstellung eines Programms ist es sinnvoll, an vielen Stellen Ausgabebefehle einzufügen, die aber bei der endgültigen Fassung entfernt werden sollen. Dafür wird gleich am Anfang das leere Makro *DEBUG* definiert.

```
#define DEBUG
...
#ifdef DEBUG
    printf("Zahl1: %d\n", Zahl1);
#endif
...
```

Wenn die zusätzlichen Ausgaben entfernt werden sollen, muss nur die Zeile *#define DEBUG* gelöscht werden.

In dem Beispiel wurde bereits der Befehl zur Beendigung der Bedingten Übersetzung verwendet:

```
#endif
```

Der Preprozessor verwendet statt der geschweiften Klammern diesen Befehl, um den bedingten Block einzugrenzen.

Wenn es darum geht zwei Programmblöcke alternativ zu übersetzen, also entweder den einen oder den anderen, so steht folgender Befehl zur Verfügung:

```
#else
```

Beispiel:

```
#ifdef DEBUG
    printf("Debug-Version\n");
#else
    printf("Release-Version\n");
#endif
```

Wie bei den Befehlen für bedingte Anweisungen gibt es beim Preprozessor die Möglichkeit Bedingungen zu verketten. Anstatt die Befehle #else und #if hintereinander zu schreiben, wird der Befehl

```
#elif
```

eingeführt. Beispiel:

```
#if MODUS==1
    printf("Modus 1\n");
#elif MODUS==2
    printf("Modus 2\n");
#elif MODUS==A
    printf("Modus A\n");
#endif
```

Die verbleibenden Befehle des Preprozessors *#line*, *#error* und *#pragma* werden hier nicht behandelt.

7.4 Quellcode auf mehrere Dateien verteilen

Soll ein größeres Programm erstellt werden, so wäre es mühsam, den gesamten Quelltext in eine Datei zu schreiben. Erstens würde die Datei unübersichtlich lang werden, zweitens könnte immer nur ein Programmierer zur Zeit daran arbeiten.

Kleine Programme können noch gut in einer Datei untergebracht werden. Mit zunehmender Größe sollte aber ein Programm auf mehrere Dateien verteilt werden. Wenn eine Datei einige hundert Zeilen hat, sollte man über eine Aufteilung nachdenken.

7.4.1 Mehrere Quellcode-Dateien übersetzen

Abbildung 7.2 zeigt das Prinzip, wie mehrere Quellcodedateien zu einem ausführbaren Programm übersetzt werden.

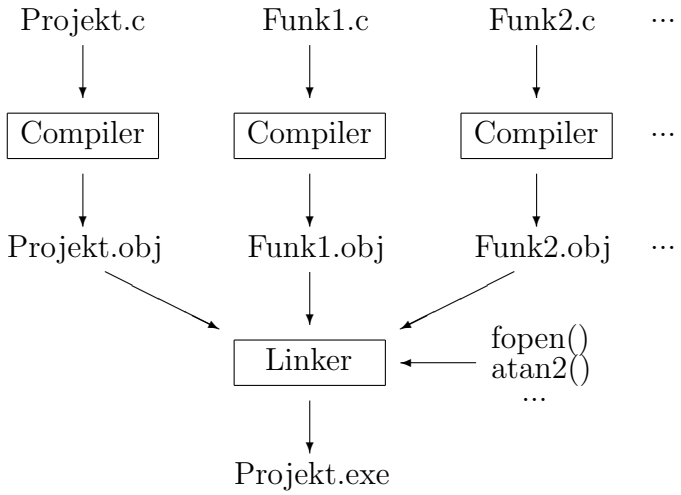


Abbildung 7.2: Übersetzung mehrerer Quellcodedateien

In einem ersten Schritt werden *Pre-Compiler* und *Compiler* (hier als *Compiler* zusammengefasst) der Reihe nach auf alle Quellcodedateien angesetzt. Für jede Quellcodedatei wird eine Objektdatei mit Endung `obj` erzeugt. Wurde ein Projekt verändert, so reicht es, die veränderten Quellcodedateien zu übersetzen.

In einem zweiten Schritt werden jetzt alle Objektdateien und die eingebundenen Bibliotheken zu einer ausführbaren Datei zusammengebunden. Dieser Schritt muss bei jeder Veränderung einer Quellcodedatei neu durchgeführt werden. Das Ergebnis ist eine ausführbare Datei mit der Endung `exe`.

7.4.2 Verknüpfung zwischen den Quellcodedateien

Sollen mehrere Quellcodes zu einem ausführbaren Programm zusammengefügt werden, so müssen einige Informationen des einen Quellcodes auch dem anderen Quellcode bekannt sein. In Abbildung 7.3 sind der Inhalt von zwei Quellcodes schemenhaft dargestellt.

Das Modul *Eingabe.c* beinhaltet einige Funktionen für die Benutzereingabe, hier die Funktionen *getInt()* und *getFloat()*. Die Funktionen stellen sicher, dass die gewünschten Informationen vom Benutzer korrekt eingelesen werden.

Damit das Hauptprogramm die Funktionen verwenden kann, muss dort die Deklaration bekannt sein. Das Bindeglied zwischen den beiden Quellcodes ist eine gemeinsame Header-Datei. In dieser Header-Datei werden die Funktionen deklariert, die beiden Quellcodes bekannt sein müssen.

Eingabe.h

```
#ifndef MODUL_EINGABE
#define MODUL_EINGABE

int getInt(char *text);
float getFloat(char *text);

#endif
```

Main.c

```
#include "Eingabe.h"

int main()
{
   int Alter;
   ...
   Alter = getInt("Alter:");
   ...
   return 0;
}
```

Eingabe.c

```
#include "Eingabe.h"

int getInt(char *text)
{
   ...
}

float getFloat(char *text)
{
   ...
}
```

Abbildung 7.3: Ein kleines Projekt

In einem Projekt werden verschiedene Module implementiert, die dann zu einem größeren Programm zusammengebunden werden. Üblicherweise wird für jedes Modul eine eigene Header-Datei erstellt. Die Programmblöcke, die mit diesem Modul arbeiten, müssen nur noch die entsprechende Header-Datei einbinden, und können dann die Funktionen des Moduls verwenden.

Beim Übersetzen eines auf mehrere Dateien verteilten Programms werden zunächst alle Quellcodedateien mit Endung .c in übersetzte Programmstücke, den Objektdateien mit Endung .obj, übersetzt. In diesen Objekt-Dateien sind an den Stellen, wo ein Aufruf zu einer Funktion außerhalb des aktuellen Moduls erfolgt, nur Markierungen eingetragen. In einem zweiten Schritt wertet der Linker diese Markierungen aus, und bindet die entsprechenden Funktionen zusammen.

Sie können auch Variablen über mehrere Dateien verteilt verwenden. Wichtig ist, dass die Variable nur an einer Stelle definiert wird; Deklarationen können beliebig wiederholt werden. Da die Headerdatei von mehreren Programmblöcken aus eingefügt wird, ist dies nicht der richtige Ort für die Definition von Variablen. Das sollte in den Programmblöcken selbst geschehen. Die Deklaration einer Variablen erfolgt durch das vorgestellte Schlüsselwort *extern*. Beispiel:

```
extern FILE *inp, *out;    /* Ein- und Ausgabedatei */
extern int errorCounter;   /* Zähler für alle Fehler */
```

7.4.3 Gestaltung einer Header-Datei

In einer Header-Datei sollten nur die Funktionen deklariert werden, die außerhalb des Moduls verwendet werden. Beispiel: Sie erstellen in einem Modul die Funktion *getFloat()*, die ihrerseits die Funktion *checkRange()* aufruft. Erstere soll von außen aufgerufen werden, letztere nur innerhalb des Moduls. Daher wird die Deklaration für *getFloat()* in die Header-Datei eingetragen, während die die Funktion *checkRange()* nur im Quellcode des Moduls erfolgt, siehe Abbildung 7.4.

Eingabe.h

```
#ifndef MODUL_EINGABE
#define MODUL_EINGABE
float getFloat(char *text, float min, float max);
#endif
```

Eingabe.c

```
#include "Eingabe.h"
int checkRange(float number, float min, float max);
float getFloat(char *text)
{
   ...
}
int checkRange(float number, float min, float max)
{
   ...
}
```

Abbildung 7.4: Lokale Deklaration einer Funktion

Bei wachsenden Projekten wird die Verknüpfung über Header-Dateien zunehmend komplexer. Das Hauptprogramm bindet Module ein, die weitere Module einbinden, die ihrerseits wieder andere Module einbinden etc. Dabei kann es passieren, dass eine Header-Datei mehrfach eingebunden wird. Damit Funktionen und Makros nicht doppelt deklariert werden, sollte jeweils die gesamte Header-Datei bedingt übersetzt werden. Das geschieht mit den folgenden drei Zeilen:

```
#ifndef EIGENER_MAKRO_NAME_DER_GERNE_LANG_SEIN_DARF
#define EIGENER_MAKRO_NAME_DER_GERNE_LANG_SEIN_DARF

/* Hier werden die Deklarationen eingefügt */

#endif /* #ifndef EIGENER_MAKRO_NAME_DER_GERNE_LANG_SEIN_DARF */
```

In der ersten Zeile wird mit *#ifndef* geprüft, ob ein bestimmter Makroname noch nicht definiert worden ist. Nur wenn es diesen Namen noch *nicht* gibt, werden die folgenden Zeilen bis zum *#endif*-Befehl ausgeführt. Als Makroname sollte ein sprechender und vor allem eindeutiger Name gewählt werden. Damit der Name eindeutig ist, kann er ruhig etwas länger ausfallen.

Als nächstes wird der Makroname definiert. Auch wenn dem Makronamen kein Inhalt zugewiesen wird, existiert er ab dieser Zeile. Alle Deklarationen innerhalb der *#ifndef*-Verzweigung werden aber noch normal übersetzt.

Wird nun dieselbe Header-Datei ein zweites mal eingebunden, so existiert der entsprechende Makroname, und die *#ifndef*-Anweisung sorgt dafür, dass die Deklarationen kein zweites mal erfolgen.

7.5 Eine Software entsteht

7.5.1 Vom Anforderungsprofil bis zur Wartung

Bei der Erstellung professioneller Software stellen die im vorigen Abschnitt beschriebenen Stufen nur eine untergeordnete Einteilung dar. Im Folgenden soll ein typischer Entstehungsprozess einer Software in kurzer Form gezeigt werden.

1. Ganz am Anfang wird ein *Anforderungsprofil* erstellt. In diesem Anforderungsprofil wird mit dem Auftraggeber abgestimmt, was genau die Software leisten soll. Die zu schreibende Software wird später an diesem Anforderungsprofil gemessen. Sie schützt aber auch den Hersteller der Software vor späten, nicht verabredeten Ergänzungswünschen des Auftraggebers.

2. Das *Grobdesign* legt die grobe Struktur der Software fest. Welche Programmblöcke soll es geben? Was sind die Aufgaben dieser Blöcke und wie sieht die Kommunitkation zwischen den Blöcken aus?

3. Im *Feindesign* wird jeder Programmblock detailliert beschrieben. Das Feindesign dient als Grundlage für das eigentliche Programmieren. Als Hilfsmittel können hier z.B. Struktogramme erstellt werden, siehe Abschnitt D.

4. Bei der *Implementierung* wird das Feindesign in den entsprechenden Programmcode übertragen. Erst hier muss die Programmiersprache beherrscht werden. Die Implementierung ist eher als ein Handwerk zu verstehen. Es gibt Programme auf dem Markt, die geeignet geschriebene Feindesigns direkt in Quellcode einer Programmiersprache übertragen.

5. In der nun folgenden *Testphase* werden alle Programmstücke systematisch getestet. Dabei wird bei allen denkbaren Programmabläufen, vor allem bei Fehlern von außen (z.B. Fehleingaben), geprüft, ob das Programm vernünftig damit umgehen kann. Oft werden für diese Tests eigene Programme geschrieben, die einen jahrelangen Betrieb der Software simulieren.

6. Anschließend wird das Programm beim Auftraggeber installiert. In dieser *Installationsphase* stellt sich heraus, ob die Software beim Auftraggeber tatsächlich genauso läuft wie in der Entwicklungsumgebung.

7. Die letzte Phase ist die *Wartungsphase*. Eine Software ist etwas lebendiges. Es gibt immer kleine Änderungen am System, auf die ggf. mit kleinen Änderungen in der Software reagiert werden muss. Aber auch Kundenwünsche gilt es zu befriedigen. Das endgültige Ende der Softwareentwicklung ist erst dann erreicht, wenn der Auftraggeber eines Tages die Software von seinem Rechner entfernt.

Diese sieben Schritte stellen idealisiert die Entwicklungsphasen einer Software dar, die der Reihe nach bearbeitet werden. Die Realität sieht etwas anders aus.

Zum einen lassen sich die Phasen nicht von einander trennen. Bei der Erstellung des Anforderungsprofils wird bereits am Grob- und Feindesign gearbeitet. Durch späte Wünsche des Kunden fließen zu einem späten Zeitpunkt immer noch weitere

Wünsche ein, die nachträglich in das Anforderungsprofil aufgenommen werden. Das heißt, die sieben Phasen gehen *fließend* ineinander über.

Zum anderen kann sich bei einer späteren Phase herausstellen, dass das Ergebnis einer früheren Phase geändert werden muss. Beispiel: Beim Testen oder der Installation treten Fehler auf, die in den vorherigen Phasen (Design und Implementierung) ausgebessert werden müssen. Das heißt, die sieben Schritte haben nicht nur eine vorwärts gerichtete Abhängigkeit, sondern frühere Schritte sind auch von späteren abhängig und haben somit eine *rückwärts* gerichtete Abhängigkeit.

Ein weiterer Aspekt, der in den genannten Phasen noch nicht zum Ausdruck kommt, ist das *Qualitätsmanagement*. Besonders bei sicherheitsrelevanter Software gibt es hohe gesetzlichen Anforderungen, die besonders behandelt werden müssen. So muss hier als Teil des *Risikomanagements* eine *Gefahrenanalyse* erstellt werden. Im Laufe des Projekts müssen alle Gefahrenpotentiale auf ein akzeptables Maß reduziert werden. Soll eine Software international vertrieben werden, so müssen zudem die Bestimmungen der jeweiligen Länder berücksichtigt werden.

7.5.2 Top-Down- und Bottom-Up-Entwurf

Der im vorherigen Abschnitt beschriebene Entstehungsprozess einer Software ist ein typischer Top-down-Entwurf. Man fängt von oben mit der Überlegung, was die Software eines Tages können soll an. Danach arbeitet man sich über das Grob- und Feindesign runter, bis man schließlich bei der Implementierung landet. Bei der Erstellung mittlerer und größerer Anwendungen ist dies der zu bevorzugende Weg.

Bei kleinen Anwendungen gibt es auch den umgekehrten Weg. Man setzt sich mit einer diffusen Idee an den Rechner, und fängt mit der Implementierung einiger Routinen ohne irgendein Schriftstück an. Im laufe der Zeit wird das Programm immer wieder umstrukturiert, bis es einen einigermaßen logischen Aufbau erhält. Das Ergebnis ist normalerweise ein schlecht wartbarer und störanfälliger Quellcode. Will man einfach mal einen neuen Algorithmus ausprobieren, kann dieser Bottom-Up-Entwurf der leichtere und vor allem schnellere Weg sein. Ansonsten ist insbesondere bei der Erstellung professioneller Software dringendst davon abzuraten.

7.6 Aufgaben

Aufgabe 7.1: Welches Programm verarbeitet Makros (*#define*), Pre-Compiler, Complier oder Linker?

Aufgabe 7.2: Welches Programm fügt die einzelnen Programmblöcke zusammen, Pre-Compiler, Complier oder Linker?

Aufgabe 7.3: Erläutern Sie beim Präprozessor-Befehl **#include** den Unterschied zwischen spitzen Klammern und Anführungszeichen.

Aufgabe 7.4: Definieren Sie ein Makro mit Namen *ANZAHL* und weisen Sie ihm die Zeichenfolge 100 zu.

Aufgabe 7.5: Definieren Sie ein Makro mit Namen *DEBUG*.

Aufgabe 7.6: Erstellen Sie ein Stück Programmcode, dass nur bei Existenz des Makros *TEST* übersetzt wird.

Aufgabe 7.7: Wozu dienen in C Header-Dateien?

Aufgabe 7.8: Wie teilt ein in eine Datei ausgelagerter Programmblock dem Rest des Programms seine Funktionsdeklarationen mit?

Aufgabe 7.9: Erstellen Sie eine Header-Datei für die Funktion **int** getInt(**char** *text). Stellen Sie dabei sicher, dass die Funktion nur einmal deklariert wird, auch wenn die Header-Datei mehrfach eingebunden wurde.

Aufgabe 7.10: Sie haben ein Programm auf drei Quellcode-Dateien verteilt. Wie oft werden Complier und Linker für eine vollständige Erstellung des Programms aufgerufen?

Aufgabe 7.11: Beschreiben Sie unter welchen Bedingungen die *top-down*- und die *bottom-up*-Methode verwendet wird.

Aufgabe 7.12: Wozu dient bei einem größeren Softwareprojekt das *Anforderungsprofil*?

Aufgabe 7.13: Was passiert bei einem größeren Softwareprojekt beim *Feindesign*?

Aufgabe 7.14: Wie lange dauert bei einem Softwareprojekt die *Wartungsphase*?

Anhang A

Einige nützliche Funktionen

A.1 Formatierte Ein- und Ausgabe

In der Standardbibliothek für die Ein- und Ausgabe, *stdio.h*, werden unter anderem die Funktionen *printf* und *scanf* bereitgestellt. Die Funktion *printf* (print formatted, zu deutsch: drucke formatiert) dient der formatierten Ausgabe aller elementaren Datentypen auf den Bildschirm. Die Funktion *scanf* (scan formatted, zu deutsch: lese formatiert) hilft beim Lesen elementarer Datentypen.

A.1.1 Die Funktion *printf*

Die Funktion *printf* dient der Ausgabe von Informationen auf dem Bildschirm. Neben einfachen Textzeilen können alle elementaren Datentypen der Sprache C formatiert ausgegeben werden.

Die Funktion *printf* erwartet einen oder mehrere Parameter. Der erste, obligatorische Parameter ist eine Zeichenkette, der die Ausgabe formatiert. Die nachfolgenden optionalen Parameter beinhalten die Daten, die dargestellt werden sollen:

printf(*Formatstring*, ...);

Der Formatstring kann eine Konstante oder eine Variable sein. Er beinhaltet normalen Text, mit einigen Platzhaltern für die Daten, die ausgegeben werden sollen. Ein Platzhalter wird mit einem Prozentzeichen '%' eingeleitet und endet mit einem Buchstaben, der den Datentyp repräsentiert.

Beispiel: Die Variable v steht für die Geschwindigkeit eines Autos und ist vom Typ *float*. Die Ausgabe der Geschwindigkeit erfolgt mit folgender Anweisung:

```
float v=57.3;
printf("Geschwindigkeit: %f km/h\n", v);
```

Auf dem Bildschirm erscheint:

```
Geschwindigkeit: 57.300000 km/h
```

Der Platzhalter für die Variable v wird hinter den Zeichen *Geschwindigkeit:* mit dem Prozentzeichen '%'eingeleitet, gefolgt von dem Buchstaben f für den Datentyp *float*.

Die Funktion *printf* kann auch mit nur einen Parameter zur reinen Textausgabe verwendet werden. Die Anweisung

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

gibt den Text *hello world* auf den Bildschirm aus.

Der Platzhalter kann neben den Buchstaben für den Datentyp auch Angaben zur Formatierung des Datums enthalten. Allgemein hat ein Platzhalter folgendes Format:

`%[`*Flags*`][`*Breite*`][.`*Präzision*`][h|l|L]` *Typ*

(Die Angaben in den eckigen Klammern sind optional.) Im Folgenden betrachten wir der Reihe nach den *Typ*, den *Modifizierer* (`h`, `l` oder `L`), die *Breite*, die *Präzision* und schließlich die *Flags*.

Formatstring: *Typ*

Der Typ gibt allgemein den Datentyp an, der ausgegeben werden soll. Die möglichen Typen sind in Tabelle A.1 zusammengefasst.

Typ	Datentyp	Darstellung
%d oder %i	signed int	dezimal
%u	unsigned int	dezimal
%o	unsigned int	oktal
%x	unsigned int	hexadezimal (mit kleinen Buchstaben)
%X	unsigned int	hexadezimal (mit großen Buchstaben)
%f	float	immer ohne Exponent
%e	float	immer mit Exponent (durch 'e' angedeutet)
%E	float	immer mit Exponent (durch 'E' angedeutet)
%g	float	nach Bedarf mit Exponent (durch 'e' angedeutet)
%G	float	nach Bedarf mit Exponent (durch 'E' angedeutet)
%c	char	einzelnes Zeichen
%s	char[]	Zeichenkette (String)
%p	void *	Speicheradresse
%n	signed int *	schreibt die Anzahl der bisherigen Zeichen an die angegebene Adresse
%%	-	Ausgabe des Zeichens '%'

Tabelle A.1: Der *Typ* im Formatstring für die elementaren Datentypen.

Beispiel: Das Programmstück

```
char c='A';
int i=2;
float f=1.23f;

printf("c: %c\n", c);
printf("i: %d\n", i);
printf("f: %f\n", f);
```

erzeugt folgende Ausgabe:

```
c: A
i: 2
f: 1.230000
```

Formatstring: *Modifizierer*

Bisher konnten wir mit dem Typ nur einige der elementaren Datentypen ansprechen. Durch den *Modifizierer* können auch die anderen Datentypen ausgegeben werden. Die möglichen Modifizierer sind in Tabelle A.2 aufgelistet.

Modifizierer	angewendet auf	Interpretation/Wirkung
h	d, i	short signed int
	o, u, x, X	short unsigned int
	n	short int*
l	d, i	long signed int
	o, u, x, X	long unsigned int
	n	long int*
	e, E, f, g, G	double (nur scanf)
L	e, E, f, g, G	long double

Tabelle A.2: Modifizierer für die Platzhalter der elementaren Datentypen.

Zum Beispiel erzeugt das Programmstück

```
long l=123456789;
short s=5;
long double d=0.123;

printf("l: %ld\n", l);
printf("s: %hd\n", s);
printf("d: %Le\n", d);
```

folgende Ausgabe:

```
l: 123456789
s: 5
d: 1.230000e-001
```

Formatstring: *Breite*

Wie der Name vermuten lässt kann mit der *Breite* die Breite der Ausgabe beeinflusst werden. Der angegebene Zahlenwert gibt die minimale Breite an. Benötigt die Ausgabe mehr Platz, so werden entsprechend mehr Zeichen ausgegeben. Wird die Breitenangabe mit einer Null begonnen, so werden führende Nullen ausgegeben.

Wird statt der Zahl ein Stern '*' eingetragen, so muss die Breite als Parameter vor der Variable angegeben werden. Diese Angabe der Breite kann auch mit einer Variablen angegeben werden. Tabelle A.3 fasst die Breitenangaben zusammen.

Als Beispiel gibt der folgende Programmcode

Breite	Wirkung
n	min. n Zeichen, Leerzeichen vorangestellt
0n	min. n Zeichen, Nullen vorangestellt
*	das nächste Argument aus der Liste ist die Breite

Tabelle A.3: Angabe für die Breite der Ausgabe.

```
int i=123;
long l=12345678;

printf("i:␣>%6d<\n", i);
printf("i:␣>%06d<\n", i);
printf("l:␣>%6ld<\n", l);
printf("i:␣>%*d<\n", 6, i);
```

die folgende Ausgabe auf dem Bildschirm aus:

```
i: >   123<
i: >000123<
l: >12345678<
i: >   123<
```

Formatstring: *Präzision*

Für die Gleitkommazahlen kann angegeben werden, mit wieviel Nachkommastellen sie ausgegeben werden sollen. Erfolgt gleichzeitig eine Breitenangabe, so schließt diese die Nachkommastellen und das Dezimaltrennungszeichen (den Punkt) mit ein. Es wird mathematisch gerundet. Tabelle A.4 fasst die Möglichkeiten zusammen:

Präzision	angewendet auf	Präzision
.n	e, E, f	n Nachkommastellen
	g, G	max. n Nachkommastellen
.0	e, E, f	kein Dezimalpunkt
nichts	e, E, f	6 Nachkommastellen
.*		Präzision aus Argument in Liste

Tabelle A.4: Angaben zur Anzahl der Nachkommastellen.

Beispiel: Die Programmzeilen

```
double d=12.34567;
int i;

printf("d:␣%.3lf\n", d);
printf("d:␣%.0lf\n", d);
for(i=0; i<=4; i++)
    printf("d:␣%.*le\n", i, d);
```

erzeugen folgende Ausgabe:

```
d: 12.346
d: 12
d: 1e+001
```

```
d: 1.2e+001
d: 1.23e+001
d: 1.235e+001
d: 1.2346e+001
```

Formatstring: *Flags*

Zum Schluss noch ein paar *Flags* zur weiteren Anpassung der Ausgabe. Die Möglichkeiten sind in Tabelle A.5 zusammengefasst.

Flags	angewendet auf	Bedeutung
nichts		rechtsbündig
-		linksbündig mit folgenden Leerzeichen
+		Vorzeichen immer ausgeben (auch +)
Leerzeichen		nur negative Vorzeichen
#	o	für Werte ≠0 wird eine '0' vorgestellt
	x, X	für Werte ≠0 wird '0x', bzw. '0X' vorgest.
	e, E, f, g, G	Dezimalpunkt immer ausgeben

Tabelle A.5: Einige weitere Angaben zum Ausgabeformat.

Zum Beispiel erzeugen die Programmzeilen

```
unsigned u=123;
int i=234;

printf("u: >%5u<\n", u);
printf("u: >%-5u<\n", u);
printf("i: %+d\n", i);
printf("u: %#o\n", u);
printf("u: %#x\n", u);
```

folgende Ausgabe:

```
u: >  123<
u: >123  <
i: +234
u: 0173
u: 0x7b
```

Formatstring: ein paar weitere Beispiele

In Tabelle A.6 sind noch ein paar weitere Beispiele zusammengetragen.

Was passiert bei falschen Platzhaltern?

Wenn für die darzustellenden Daten falsche Platzhalter verwendet werden kann es zu Problemen kommen. Betrachten Sie folgendes Programmstück:

```
double d=5.2;
int i=3;
printf("%d  %d\n", d, i);
```

printf-Anweisung	**Ausgabe**
float v=87.3; printf("Speed:␣%f\n", v);	`Speed: 87.300000`
printf("Das␣kostet␣%.2f\n", 15.9);	`Das kostet 15.90`
char Name[]="Meier"; **char** Titel[]="Frau␣Dr."; printf("Guten␣Tag\n%s␣%s\n", Titel, Name);	`Guten Tag` `Frau Dr. Meier`
int Zahl=12; printf("12345678\n%4d\n", Zahl);	`12345678` `  12`
printf("%.2e\n", 124.8);	`1.25e+002`
printf("%#.0f\n", 12.3);	`12.`
double Zahl=12.3456; **int** n=2; printf("12345678\n%8.*lf\n", n, Zahl);	`12345678` `12.35`

Tabelle A.6: Einige Beispiele für die Verwendung der printf-Funktion.

Beim Aufruf der Funktion *printf* werden alle Parameter der Reihe nach in den Speicher kopiert. In dem Beispiel sieht das so aus:

Abbildung A.1: Anordnung der Aufrufparameter im Speicher.

Der Platzhalter %d wird fälschlicherweise auf die Variable *d* vom Typ *double* angewendet. Das heißt, die Variable *d* wird hier als *int* interpretiert. Die Programmierumgebung, in der das Programm getestet wurde, verwendet für *double* 8 Byte und für *int* nur 4 Byte. Der Befehl *printf* nimmt daher von der Variable *d* nur die ersten vier Byte und interpretiert sie als *int*.

Der nächste Platzhalter steht wieder für den Typ *int*. Da aber der vorherige Platzhalter die Variable vom Typ *double* nur zur Hälfte eingelesen hat, versucht jetzt der zweite Platzhalter die zweite Hälfte der Variable *d* als *int* zu interpretieren. Es ist offensichtlich, das dabei Unsinn heraus kommt. In der Testumgebung sah das so aus:

```
-858993459 1075104972
```

Daher gilt: Es ist peinlichst darauf zu achten, dass die Platzhalter mit den dazugehörigen Datentypen zusammenpassen.

Rückgabewert von *printf*

Bei Erfolg gibt die Funktion die Anzahl der ausgegebenen Zeichen zurück. Tritt ein Fehler auf, so wird ein negativer Wert zurückgegeben.

A.1.2 Die Funktion *scanf*

Mit der Funktion *scanf* können Informationen vom Benutzer des Programms abgefragt werden. Der Benutzer betätigt Tasten, die in Form von Zahlen im Rechner

bearbeitet werden. Die Funktion *scanf* fügt die Codes zusammen und erstellt daraus eine sinnvolle Information. Soll vom Benutzer ein Wort erfragt werden, so müssen nur die Codes der einzelnen Tasten in Zeichen umgewandelt und zusammengefügt werden. Handelt es sich dagegen um Zahlen, so müssen die einzelnen Tastendrücke in eine sinnvolle Zahl umgewandelt werden.

Die Funktion *scanf* ist in der Lage, alle elementaren Datentypen der Programmiersprache C zu verarbeiten. Die Funktion erwartet als Parameter eine Zeichenkette mit Informationen über die Datentypen, sowie eine Liste von Zeigern, die auf die zu beschreibenden Variablen zeigen. Allgemein sieht das so aus:

scanf(*Formatstring*, ...);

Nehmen wir an, es soll eine Zahl vom Typ *int* vom Benutzer abgefragt werden, und das Ergebnis soll in die Variable *a* geschrieben werden, so sieht das folgendermaßen aus:

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

Wichtig ist der Operator '&', welcher die Adresse der Variable ermittelt. Erst dadurch ist die Funktion in der Lage, den Inhalt der Variable zu verändern.

Der Formatstring sieht dem der printf-Funktion sehr ähnlich.

`%[h|l|L]` *Typ*

Ein Platzhalter wird durch das Prozentzeichen '%' eingeleitet. Der Datentyp ergibt sich aus Typ (siehe Tabelle A.1) und Modifizierer (siehe Tabelle A.2).

Hier ein paar Beispiele:

Befehl	**Wirkung**
int i; scanf("%d", &i);	Es wird eine Variable vom Typ *int* abgefragt und in die Variable *i* geschrieben.
double d; scanf("%lf", &d);	Es wird eine Zahl vom Typ *double* abgefragt und in die Variable *d* geschrieben.
float x, y; scanf("%f␣%f", &x, &y);	Es werden zwei Zahlen vom Typ *float* abgefragt und in die Variablen x und y gespeichert.

Tabelle A.7: Einige Beispiele für die Verwendung der scanf-Funktion.

Als Rückgabewert liefert die Funktion scanf die Anzahl der fehlerfrei eingelesenen Variablen. Diese Eigenschaft hilft beim Abfangen von Fehleingaben.

A.2 Zeichenketten verarbeiten

A.2.1 strlen()

Die Funktion strlen() ermittelt die Länge einer mit null abgeschlossenen Zeichenkette. Die Syntax lautet:

size_t strlen(**const char** *text);

Headerdatei: <string.h>

text. Zeiger auf eine mit null abgeschlossene Zeichenkette.

Rückgabewert. Liefert die Länge der Zeichenkette. Das sind alle Zeichen bis zum ersten Zeichen mit dem ASCII-Code null, wobei die null selbst nicht mitgezählt wird. Der Datentyp size_t ist eine ganze Zahl und kann direkt in den Datentyp **int** kopiert werden.

A.2.2 strcpy()

Die Funktion strcpy() kopiert eine mit null abgeschlossene Zeichenkette. Die Syntax lautet:

```
char *strcpy(char *Ziel, const char *Quelle);
```

Headerdatei: <string.h>

Ziel. Zeiger auf eine mit null abgeschlossene Zeichenkette, in die der Text kopiert werden soll. Der Programmierer muss selbst auf die maximale Textlänge achten.

Quelle. Zeiger auf eine mit null abgeschlossene Zeichenkette, aus der kopiert werden soll.

Rückgabewert. Zeiger auf den kopierten Text. Das ist der selbe Wert den der Parameter Ziel hat. Für einen möglichen Fehler ist kein spezieller Rückgabewert vorgesehen.

A.2.3 strcat()

Die Funktion strcat() fügt eine mit null abgeschlossene Zeichenkette hinter eine andere mit null abgeschlossene Zeichenkette ein. Die Syntax lautet:

```
char *strcat(char *Ziel, const char *Quelle);
```

Headerdatei: <string.h>

Ziel. Zeiger auf eine mit null abgeschlossene Zeichenkette, an die der Text angehängt werden soll. Der Programmierer muss selbst auf die maximale Textlänge achten.

Quelle. Zeiger auf eine mit null abgeschlossene Zeichenkette, die an die andere Zeichenkette angehängt werden soll.

Rückgabewert. Zeiger auf den kopierten Text. Das ist der selbe Wert den der Parameter Ziel hat. Für einen möglichen Fehler ist kein spezieller Rückgabewert vorgesehen.

A.3 Mathematische Funktionen

A.3.1 exp()

Die Exponentialfunktion exp() berechnet den Exponentialwert zum übergebenen Exponenten zur Basis $e = 2.71828...$, $y = e^x$. Die Syntax lautet:

```
double exp(double x);
```

Headerdatei: <math.h>

x. Exponent der Exponentialfunktion.

Rückgabewert. Exponentialwert zur Basis e. Ist das Ergebnis zu klein, wird der Wert null zurückgegeben. Ist das Ergebnis zu groß, so beträgt der Rückgabewert INF, ein spezieller Zustand des Typs **double**.

A.3.2 pow()

Die Funktion pow() berechnet allgemein die Potenz $z = x^y$. Die Syntax lautet:

```
double pow(double x, double y);
```

Headerdatei: <math.h>

x. Basis der Potenz.

y. Exponent der Potenz.

Rückgabewert. Potenz zur Basis x und Exponenten y. Ist das Ergebnis zu klein, wird der Wert null zurückgegeben. Ist das Ergebnis zu groß, so beträgt der Rückgabewert INF, ein spezieller Zustand des Typs **double**.

A.3.3 sqrt()

Die Funktion sqrt() berechnet die Quadratwurzel. Die Syntax lautet:

```
double sqrt(double x);
```

Headerdatei: <math.h>

x. Radikand der Quadratwurzel.

Rückgabewert. Quadratwurzel des Radikanden x. Ist der Radikand negativ, so nimmt der zurückgegebene Wert einen Fehlerzustand ein.

A.3.4 log(), log10

Die Funktionen log() und log10 berechnen den natürlichen Logarithmus (zur Basis $e = 2.71828...$) und den Zehner-Logarithmus (zur Basis zehn). Die Syntax lautet:

```
double log(double x);
double log10(double x);
```

Headerdatei: <math.h>

x. Potenz, von dem der Logarithmus berechnet werden soll. Der Wert muss größer als null sein, sonst entsteht ein Fehler.

Rückgabewert. Logarithmus der übergebenen Potenz x; bei log() zur Basis $e = 2,71828...$ und bei log10() zur Basis 10. Ist die übergebene Potenz kleiner oder gleich null, so nimmt der zurückgegebene Wert einen Fehlerzustand ein.

A.3.5 sin(), cos(), tan()

Die Funktionen sin(), cos() und tan() berechnen den Sinus, Kosinus und Tangens zum übergebenen Winkel. Die Syntax lautet:

```
double sin(double x);
double cos(double x);
double tan(double x);
```

Headerdatei: <math.h>

x. Winkel, auf dem die trigonometrischen Funktion angewendet werden soll. Die Winkel werden in Bogenmaß angegeben.

Rückgabewert. Sinus, Kosinus, bzw. Tangens zum übergebenen Winkel. Bei sehr großen Winkeln (z.B. Betrag größer als 10^{20}) werden die Ergebnisse ungenau.

A.3.6 asin(), acos(), atan()

Die Funktionen asin(), acos() und atan() berechnen den Arkussinus, Arkuskosinus und Arkustangens zum übergebenen Winkel. Die Syntax lautet:

```
double asin(double x);
double acos(double x);
double atan(double x);
```

Headerdatei: <math.h>

x. Wert, auf den die Arkusfunktionen angewendet werden. Für asin() und acos() muss der Wertebereich von $-1 \leq x \leq 1$ beachtet werden.

Rückgabewert. Der berechnete Winkel in Bogenmaß. Wurde bei asin() oder acos() der Wertebereich missachtet, so nimmt der zurückgegebene Wert einen Fehlerzustand ein.

A.3.7 atan2()

Die Funktion atan2() berechnet Arkustangens von $\frac{y}{x}$ über alle vier Quadranten. Die Syntax lautet:

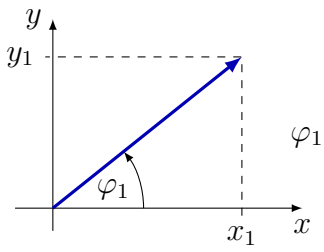
```
double atan2(double y, double x);
```

Headerdatei: <math.h>

y. Beliebiger Zahlenwert auf der Y-Achse.

x. Beliebiger Zahlenwert auf der X-Achse.

Rückgabewert. Arkustangens von $\frac{y}{x}$ im Wertebereich $-\pi < \varphi \leq \pi$. Haben x und y beide den Wert null ist das Ergebnis ebenfalls null.



$$\varphi_1 = \arctan\left(\frac{y_1}{x_1}\right)$$

Anhang B

ASCII-Tabelle

Dez/Hex/Okt	Zeichen	Dez/Hex/Okt	Zeichen	Dez/Hex/Okt	Zeichen	Dez/Hex/Okt	Zeichen
0/00/000	NUL	32/20/040	SP	64/40/100	@	96/60/140	‘
1/01/001	SOH	33/21/041	!	65/41/101	A	97/61/141	a
2/02/002	STX	34/22/042	“	66/42/102	B	98/62/142	b
3/03/003	ETX	35/23/043	#	67/43/103	C	99/63/143	c
4/04/004	EOT	36/24/044	$	68/44/104	D	100/64/144	d
5/05/005	ENQ	37/25/045	%	69/45/105	E	101/65/145	e
6/06/006	ACK	38/26/046	&	70/46/106	F	102/66/146	f
7/07/007	BEL	39/27/047	’	71/47/107	G	103/67/147	g
8/08/010	BS	40/28/050	(	72/48/110	H	104/68/150	h
9/09/011	TAB	41/29/051	)	73/49/111	I	105/69/151	i
10/0A/012	LF	42/2A/052	*	74/4A/112	J	106/6A/152	j
11/0B/013	VT	43/2B/053	+	75/4B/113	K	107/6B/153	k
12/0C/014	FF	44/2C/054	,	76/4C/114	L	108/6C/154	l
13/0D/015	CR	45/2D/055	-	77/4D/115	M	109/6D/155	m
14/0E/016	SO	46/2E/056	.	78/4E/116	N	110/6E/156	n
15/0F/017	SI	47/2F/057	/	79/4F/117	O	111/6F/157	o
16/10/020	DLE	48/30/060	0	80/50/120	P	112/70/160	p
17/11/021	DC1	49/31/061	1	81/51/121	Q	113/71/161	q
18/12/022	DC2	50/32/062	2	82/52/122	R	114/72/162	r
19/13/023	DC3	51/33/063	3	83/53/123	S	115/73/163	s
20/14/024	DC4	52/34/064	4	84/54/124	T	116/74/164	t
21/15/025	NAK	53/35/065	5	85/55/125	U	117/75/165	u
22/16/026	SYN	54/36/066	6	86/56/126	V	118/76/166	v
23/17/027	ETB	55/37/067	7	87/57/127	W	119/77/167	w
24/18/030	CAN	56/38/070	8	88/58/130	X	120/78/170	x
25/19/031	EM	57/39/071	9	89/59/131	Y	121/79/171	y
26/1A/032	SUB	58/3A/072	:	90/5A/132	Z	122/7A/172	z
27/1B/033	ESC	59/3B/073	;	91/5B/133	[	123/7B/173	{
28/1C/034	FS	60/3C/074	<	92/5C/134	\	124/7C/174	|
29/1D/035	GS	61/3D/075	=	93/5D/135	]	125/7D/175	}
30/1E/036	RS	62/3E/076	>	94/5E/136	^	126/7E/176	~
31/1F/037	US	63/3F/077	?	95/5F/137	_	127/7F/177	DEL

Tabelle B.1: Der ASCII-Zeichensatz

Anhang C

Eine "unvollständige" Einführung in Aktivitätsdiagramme

C.1 Einführung

Zur Visualisierung von Programmen (Aktivitäten) bietet die *Unified Modeling Language*, *UML*, als ein Sprachmittel die *Aktivitätsdiagramme*. Mit ihnen lassen sich Aktionen, Verzweigungen, Schleifen etc. darstellen und bieten einen grafischen Überblick über die Abläufe der Aktivität.

Warum eine *unvollständige* Einführung? Diese Seiten richten sich an Studenten der Informations- und Elektrotechnik an der Hochschule für angewandte Wissenschaften in den ersten zwei Semestern. Dort verwenden wir Aktivitätsdiagramme ausschließlich zur Planung und Darstellung von recht begrenzten Programmen. Wir wollen weder parallele Prozesse, noch objektorientierte Programme visualisieren. Unter dem Motto 'Weniger ist mehr' sollen von daher nur die benötigten Elemente von Aktivitätsdiagrammen behandelt werden.

Beim Übergang von UML 1.x zu UML 2.x haben sich die Bedeutungen der Elemente von Aktivitätsdiagrammen z.T. verändert. Die hier gegebene Einführung verwendet die neueren Definitionen der UML 2.x.

C.2 Beschreibung der wichtigsten Elemente

C.2.1 Anfang und Ende eines Programms

Der *Anfangsknoten* bildet den Anfang einer Aktivität (eines Programms) und wird durch einen ausgefüllten Punkt dargestellt: ●

Der *Endknoten* bildet das Ende einer Aktivität (eines Programms) und wird durch einen ausgefüllten Punkt mit Doppellinie angezeigt: ◉

Ein Aktivitätsdiagramm darf nur einen Anfangskonten (●), dafür aber mehrere Endknoten (◉ ◉ ◉ ...) haben.

C.2.2 Aktionen

Eine Aktion wird durch ein Rechteck mit abgerundeten Ecken dargestellt: (Aktion)

Der Inhalt einer Aktion kann sehr unterschiedlich ausfallen. 'Überschrift ausgeben', 'Tastendruck abfragen', 'Variable i inkrementieren' sind feingliedrige Aktionen.

'100 Primzahlen berechnen', 'Mail senden', 'Datei verschlüsseln' dienen einer gröberen Einteilung.

In den Grundsemestern werden wir unsere Aktivitätsdiagramme eher fein gliedern.

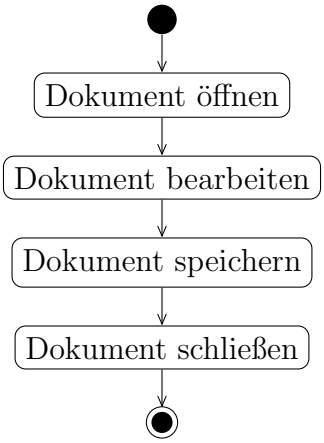
C.2.3 Kontrollfluss

Die Elemente eines Aktivitätsdiagramms werden mit Kanten in Form von Pfeilen (⟶) verbunden. Die Reihenfolge der Aktionen erfolgt in Richtung der Pfeile.

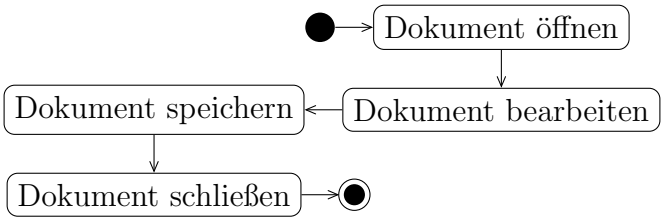
Eine kleine Aktivität mit nur einer Aktion sieht damit so aus:

Grundsätzlich können die Aktionen und Kannten beliebig angeordnet werden. Für eine gute Lesbarkeit sollten die Aktivitäten aber möglichst links oder oben beginnen und rechts oder unten enden.

Beispiel: Sie bearbeiten ein bereits existierendes Dokument und wollen das in Form eines Aktivitätsdiagramms darstellen:

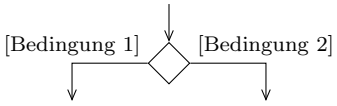


Sie können die selbe Aktivität auch etwas kompakter zu Papier bringen:

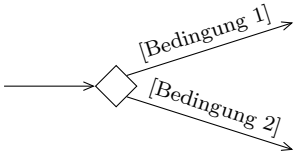


C.2.4 Verzweigung und Zusammenführung

Eine Verzweigung wird durch eine Raute dargestellt, an die *eine* Kante endet, und von der *mehrere* Kanten ausgehen. Die Bedingungen für die Entscheidung, über welche der Kanten die Raute verlassen wird, wird in kleinen Texten in eckigen Klammern angegeben:

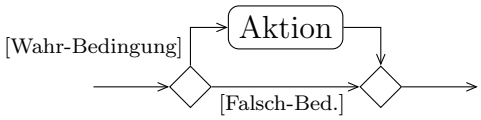


Ob dabei wie hier mit rechten Winkeln oder wie unten etwas freier mit Diagonalen gearbeitet wird, ist dem Schreiber des Aktivitätsdiagramms überlassen. Im Vordergrund sollte immer die Lesbarkeit stehen.

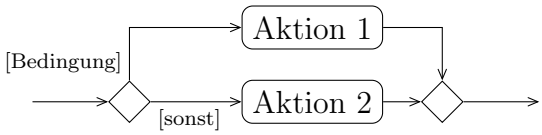


Die Bedingungen in den eckigen Klammern müssen *vollständig* und *widerspruchsfrei* sein. Das Gegenstück zu einer Verzweigung ist die Zusammenführung, für das wir wieder die Raute verwenden:

Eine bedingte Aktion (*if*) wird wie folgt dargestellt:

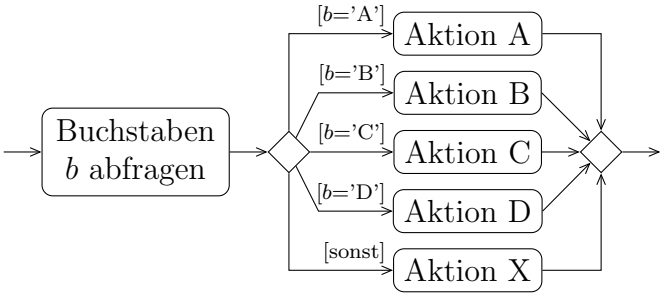


Oft genügt es für die *Falsch-Bedingung* einfach [*sonst*] oder [*else*] zu schreiben. Eine Entscheidung zwischen zwei Aktionen (*if-else*) ergibt sich mit:

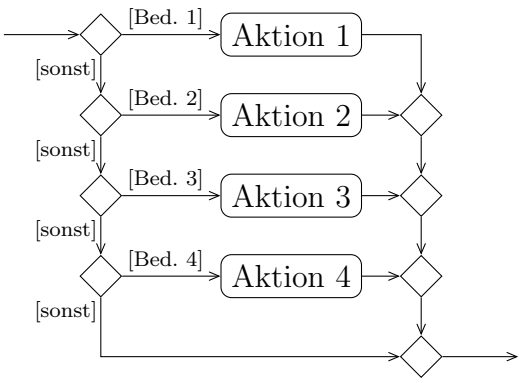


Entsprechend kann zwischen mehr als zwei Alternativen entschieden werden (*switch-case*).

Beispiel: Vom Benutzer wird ein Buchstabe (Tastendruck) abgefragt und es soll in Abhängigkeit des gewählten Buchstabens verzweigt werden:

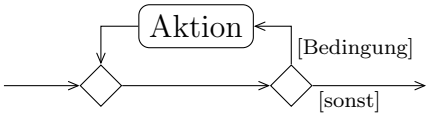


Eine verkettete Verzweigung (*else-if*) ergibt sich mit:



C.2.5 Schleifen

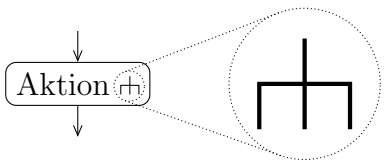
Bei einer kopfgesteuerten Schleife (*while* und *for* in ANSI C) findet die Prüfung der Schleifenbedingung durch eine Verzweigung *vor* dem Inhalt der Schleife statt:



Bei einer fußgesteuerten Schleife (*do-while* in ANSI C) findet die Prüfung der Schleifenbedingung *nach* dem Inhalt der Schleife durch eine Verzweigung statt:

C.2.6 Unterprogramme

Unterprogramme werden wie eine Aktion dargestellt, bei der rechts eine Art Gabel (⫪) eingetragen wird.



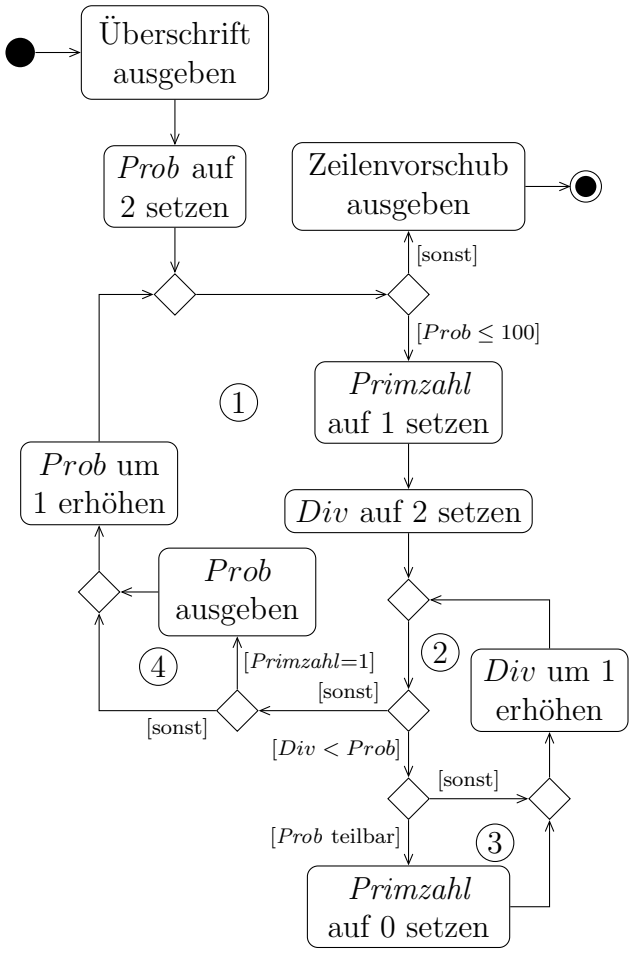
Für jeden Aufruf eines Unterprogramms muss ein Aktivitätsdiagramm mit demselben Namen existieren.

C.3 Weitere Elemente

Es gibt noch weitere Elemente für Aktivitätsdiagramme. So können für *parallele Prozesse* Aktionen parallelisiert und wieder synchronisiert werden. Des Weiteren können neben den Aktionen in Rechtecken mit runden Ecken auch *Daten* in Rechtecken mit normalen Ecken angegeben werden. Der interessierte Leser wird im Internet und in der Bibliothek fündig.

C.4 Ein Beispiel

Als Beispiel soll hier ein kleines Programm dienen, welches alle Primzahlen bis 100 berechnet:



Das Programm (die Aktivität) hat eine Äußere ① und eine innere ② Schleife. In beiden Fällen handelt es sich um kopfgesteuerte Schleifen, die sich in C mit den Befehlen *while* oder *for* realisieren lassen.

Des Weiteren finden wir zwei bedingte Aktionen: Bei ③ wird geprüft, ob der Proband durch den Divisor teilbar ist, und wenn ja, wird der Marker für Primzahl zurückgesetzt. Bei ④ wird geprüft, ob der Marker für Primzahl gesetzt ist, und wenn ja, wird die Primzahl ausgegeben.

Eine mögliche Implementierung sieht wie folgt aus:

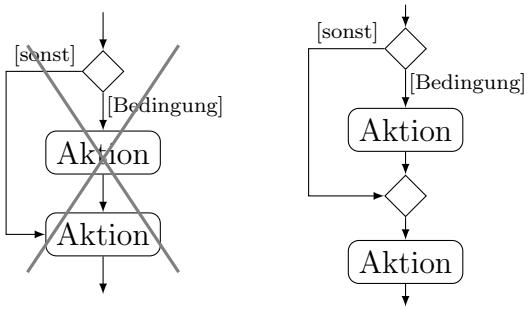
```
/* Überschrift */
printf("Primzahlen bis 100:\n");
/* Schleife über alle Probanden */
Prob = 2;
while(Prob<=100) {
    /* Annahme: Primzahl */
    Primzahl = 1;
    /* Schleife für alle Divisoren */
    for(Div=2; Div<Prob; Div++) {
        /* Wenn teilbar */
```

```
        if(Prob%Div==0)
            Primzahl = 0;
    }
    /* Ggf. Primzahl ausgeben */
    if(Primzahl)
        printf("%8d", Prob);
    Prob++;
}
printf("\n"); /* Zeilenvorschub */
```

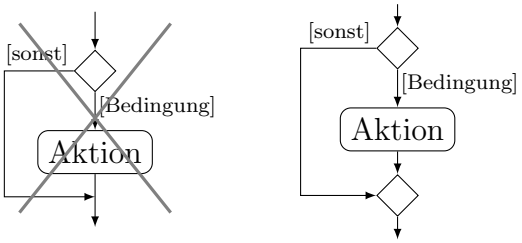
C.5 Typische Fehler

Abschließend sollen hier ein paar typische Fehler gezeigt werden:

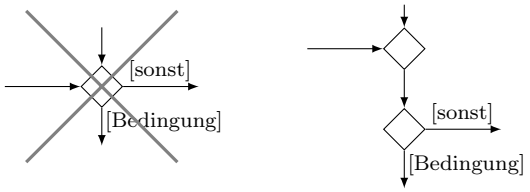
- Aktionen dürfen nicht gleichzeitig der Zusammenführung dienen:



- Anders als bei Flussdiagrammen erfolgt eine Zusammenführung immer über eine Raute:



- Zusammenführung und Verzweigung sollen getrennt werden:



Anhang D

Struktogramme

In den letzten Abschnitten wurden *Aktivitätsdiagramme* zur Veranschaulichung herangezogen. Sie sind intuitiv lesbar und bieten alle Freiheiten beim Aufbau eines Programms. Gerade diese Freiheit kann dem Aktivitätsdiagramm zum Verhängnis werden: Die Programme werden mit unter sehr kompliziert.

Eine Alternative zum Aktivitätsdiagramm ist das *Struktogramm*. Die einzelnen Elemente bestehen ähnlich wie bei Aktivitätsdiagrammen aus Blöcken, nur dass diese ohne Pfeile direkt aneinander gefügt werden.

Die Eingänge eines Blocks befinden sich grundsätzlich an der oberen Kante, die Ausgänge grundsätzlich an der unteren Kante. Seitlich gibt es weder Ein- noch Ausgänge. Im folgenden werden die einzelnen Blöcke erläutert.

D.1 Anweisungen und Blöcke

Eine Sequenz von Anweisungen wird durch untereinander angeordnete Blöcke zu einem Block zusammengefasst.

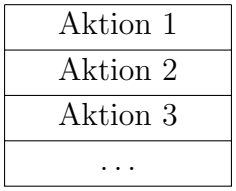


Abbildung D.1: Struktogramm einer Sequenz von Anweisungen.

D.2 Verzweigungen

D.2.1 Bedingte Verarbeitung (if-Verzweigung)

Bei der bedingten Verarbeitung, siehe Abbildung D.2, wird in den Keil die Bedingung geschrieben, auf der Unterseite geht es mit zwei Blöcken weiter.

In diesem Fall bleibt der zweite Block leer. Nur wenn die Bedingung zutrifft, wird der linke Block ausgeführt. Trifft die Bedingung nicht zu, so wird der leere Block, also nichts, ausgeführt.

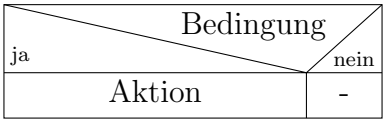


Abbildung D.2: Struktogramm einer bedingten Verarbeitung (if).

D.2.2 Einfache Alternative (if-else-Verzweigung)

Die einfache Verzweigung ähnelt sich stark der bedingten Verarbeitung, nur dass der rechte Block unter der Bedingung jetzt auch Anweisungen enthält.

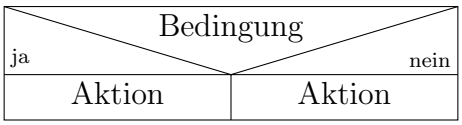


Abbildung D.3: Struktogramm einer einfachen Alternative (if-else).

D.2.3 Verkettete Verzweigungen (else-if-Verzweigung)

Die verkettete Verzweigung lässt sich durch eine verschachtelte Verzweigung darstellen, in der sich der Nein-Zweig weiter aufteilt, siehe Abbildung D.4.

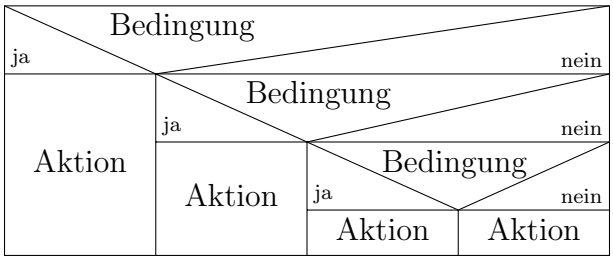


Abbildung D.4: Struktogramm einer verketteten Verzweigung (else-if).

D.2.4 Mehrfache Alternative (switch-Verzweigung)

Die mehrfache Alternative bietet statt der zwei Möglichkeiten mehrere Alternativen unter dem Keil mit der Bedingung, siehe Abbildung D.5.

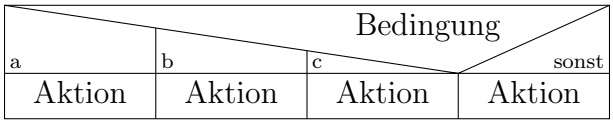


Abbildung D.5: Struktogramm einer mehrfachen Alternative (switch).

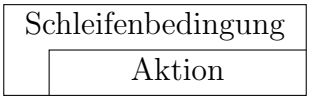


Abbildung D.6: Struktogramm einer kopfgesteuerten Schleife (while).

D.3 Schleifen

D.3.1 Kopfgesteuerte Schleifen (while und for)

Bei einer kopfgesteuerten Schleife wird die Schleifenbedingung sinngemäß vor die Anweisung(en) der Schleife geschrieben, siehe Abbildung D.6. Die Initialisierung und das Inkrement einer *for*-Schliefe muss mit separaten Anweisungen ausgedrückt werden, siehe Abbildung D.7.

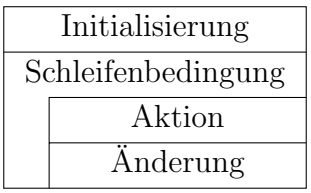


Abbildung D.7: Struktogramm einer kopfgesteuerten Schleife (for).

D.3.2 Fußgesteuerte Schleife (do)

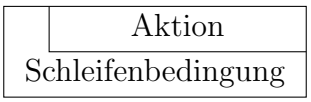


Abbildung D.8: Struktogramm einer fußgesteuerten Schleife (do).

Bei der fußgesteuerten Schleife wird die Schleifenbedingung sinngemäß hinter die Anweisung(en) eingetragen, siehe Abbildung D.8.

D.3.3 Unterbrechung von Schleifen (break und continue)

Die Unterbrechung einer Schleife kann wie in Abbildung D.9 dargestellt angedeutet werden.

Abbildung D.9: Struktogrammelement zur Unterbrechung von Schleifen.

Der Kommentar muss deutlich machen, warum und wohin unterbrochen wird, bzw. ob die Unterbrechung mit oder ohne Überprüfung der Schleifenbedingung erfolgt (*continue* oder *break*). Das gleiche Symbol wird für den Abbruch eines Unterprogramms verwendet.

Mit dieser Option bietet das Struktogramm eine Möglichkeit, die dem Ansatz der strukturierten Programmierung widerspricht. Der Gedanke, dass ein Programm immer von oben nach unten durchlaufen wird, und dass dadurch das Programm nachvollziehbar bleibt, wird durch die Unterbrechung an beliebigen Stellen stark gestört. Viele Einführungen zu dem Thema *Struktogramme* lassen diese Option gänzlich aus.

Von daher wird auch hier empfohlen, diese Option nicht zu verwenden!

D.3.4 Absolute Sprünge (goto und Marken)

Absolute Sprünge sind in der strukturierten Programmierung nicht zulässig. Daher lassen sich diese in Struktogrammen nicht darstellen.

D.4 Unterprogramme

D.4.1 Aufruf eines Unterprogramms (Funktionen)

Der Aufruf von Unterprogrammen, in C spricht man von Funktionen, sieht wie in Abbildung D.10 aus. (Funktionen in C werden im Abschnitt 5.1 auf Seite 52 behandelt.)

Abbildung D.10: Struktogrammelement zum Aufruf von Unterprogrammen.

D.4.2 Vorzeitiges Beenden eines Unterprogramms

Siehe Beeinflussung von Schleifen in Abschnitt D.3.3.

D.5 Beispiel eines Struktogramms

Als Beispiel soll ein einfacher Primzahlgenerator dienen, der die Primzahlen bis 100 ermittelt.

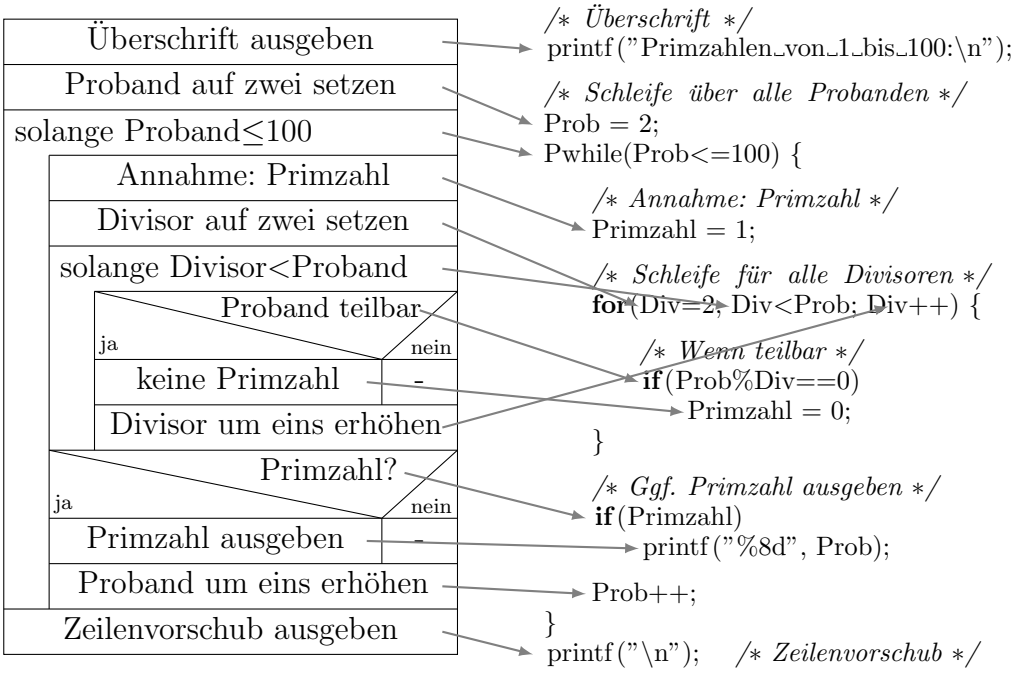


Abbildung D.11: Beispiel eines Struktogramms inkl. Quellcode.

Anhang E

Lösungen zu den Aufgaben

Lösungen zu Kapitel 1

1. a) $1\,0111_2$ b) $10\,1101_2$ c) $1000\,0001_2$ d) $0{,}1_2$ e) $11{,}11_2$ f) $1\,0001{,}0001_2$
   g) $110\,0011{,}1\overline{1100}_2$ h) $0{,}\overline{00011}_2$ i) $0{,}\overline{0000\,0010\,1000\,1111\,0101\,11}_2$

2. a) 31_{10} b) 153_{10} c) 375_{10} d) $0{,}5_{10}$ e) $0{,}125_{10}$ f) $0{,}0625_{10}$

3. a) $1111\,1011_2$ b) $1111\,1111_2$ c) $1100\,0000_2$

4. a) 212_3 b) 100_7 c) $0{,}\overline{1}_3$ d) $0{,}1_7$

5. Ein Programm in *Maschinensprache* ist für den Menschen schwer zu lesen, wird aber von der CPU eines Computers unmittelbar verstanden. Ein Programm, das in einer *höheren Programmiersprache* geschrieben wurde, ist für den Menschen besser lesbar, muss aber für den Computer vorher in die passende Maschinensprache übersetzt werden.

6. Siehe Tabelle 1.3.

Lösungen zu Kapitel 3

1. Aus großen und kleinen Buchstaben, den Ziffern 0 bis 9 und dem Unterstrich.

2. 1Zahl, **break**, Zahl 1, **for**, 2_Noten

3. **int** Zahl;

4. **unsigned long** Nummer;

5. **double** Zahl;

6. **float**, **double** und **long double**

7. a) 12, **long** b) 10, **int** c) 18, **int** d) 10, **long** e) 1, **unsigned**
   f) 186, **unsigned long**

8. a) Zeilenvorschub b) Wagenrücklauf c) Tabulator d) “backspace”
   e) Signalton f) Gegenstrich

9. a) 14 b) 2 c) 2 d) 2 e) 3 f) 0 g) 3 h) 2.4 i) 3.6

10. a) 1 (wahr) b) 1 (wahr) c) 1 (wahr) d) 0 (falsch) e) 1 (wahr) f) 0 (falsch)

11. a) 0 (falsch), 1, 0 b) 1 (wahr), 1, 0 c) 1 (wahr), 2, 1 d) 0 (falsch), 0, 1 e) 3, 2, 2 f) 1, 0, 0

12. a) 3 b) 7 c) 8 d) 3 e) 6 f) -1

13. a) 2 b) -2 c) 2

Lösungen zu Kapitel 4

1. Mit geschweiften Klammern.

2. `if(i<0) i = -i;`

3.
```
if(i<0) printf("negativ");
else printf("nicht negativ");
```

4.
```
switch(i) {
case 1:
    printf("Gabel");
    break;
case 2:
    printf("Messer");
    break;
case 3:
    printf("Essloeffel");
    break;
case 4:
    printf("Teeloeffel");
    break;
}
```

5. Eine *fußgesteuerte* Schleife prüft die Bedingung der Schleife am Ende eines Schleifendurchgangs und wird von daher mindestens einmal ausgeführt. Eine *kopfgesteuerte* Schleife prüft die Bedingung zu Beginn eines Schleifendurchgangs und wird von daher ggf. kein einziges mal ausgeführt.

6.
```
int i;
for(i=0; i<10; i++) printf("Hallo\n");
```

7.
```
int i;
for(i=1; i<=20; i++) printf("%d*%d = %d\n", i, i, i*i);
```

8.

```
float a=1;
while(a!=0) {
    printf("a = %g\n", a);
    a /= 2;
}
```

9. Absolute Sprünge machen ein Programm *unübersichtlich* und entsprechen nicht der *strukturierten Programmierung*.

Lösungen zu Kapitel 5

1.

```
void hello()
{
    printf("hello world\n");
}
```

2. **double** Betrag(**double** x, **double** y);

3. Indem der Funktion die Adressen der Variablen übergeben wird.

4.

```
void swapDouble(double *a, double *b)
{
    double tmp;
    tmp = *a;
    *a = *b;
    *b = tmp;
}
```

5. *Globale* Variablen werden außerhalb von Funktionen definiert und können von allen Teilen eines Programms gelesen und verändert werden. *Lokale* Variablen werden innerhalb einer Funktion definiert und können nur innerhalb dieser Funktion gelesen und verändert werden.

6. Da globale Variablen von allen Teilen eines Programms gelesen und verändert werden können, stellen sie eine unnötige Fehlerquelle dar. Tritt ein Fehler im Kontext einer globalen Varibale auf, so muss der ganze Quelltext nach der Fehlerursache untersucht werden!

Lösungen zu Kapitel 6

1. **double** Liste[10];

2.

```
int i;
for(i=0; i<10; i++)
    Liste[i] = 0;
```

3. Liste[9] = 3;

4. Ausgehend von 8 Byte für den Datentyp *double* benötigt der Verktor 10*8 = 80 Byte im Speicher.

5. `int prime[] = { 1, 2, 3, 5, 7, 11 };`

6. Es wird für drei Stunden á 60 Minuten á 60 Sekunden alle 10 Sekunden ein Messpunkt gespeichert. Es wird demnach für 3*60*60/10+1=1081 Messpunkte Speicher benötigt. Der zusätzliche Punkt ist nötig, damit auch der erste und letzte Messpunkt gespeichert werden kann: `double Temperatur[1081];`

7. Im Gegensatz zu den elementaren Datentypen wird bei der Übergabe eines Vektors an eine Funktion keine Kopie erstellt und die Funktion kann den Vektor der aufrufenden Funktion direkt manipulieren.

8. `int Feld [8][8];`

9.
```
int i, j;
for(i=0; i<8; i++)
    for(j=0; j<8; j++)
        Feld[i][j] = 0;
```

10.
```
Feld[0][0] = 1;
Feld[0][7] = 1;
Feld[7][0] = 1;
Feld[7][7] = 1;
```

11. Ausgehend von 4 Byte für den Datentyp *int* benötigt der Vektor 8*8*4 = 256 Byte im Speicher.

12.
```
float data[100][100][100];
int i, j, k;

for(i=0; i<100; i++)
    for(j=0; j<100; j++)
        for(k=0; k<100; k++)
            data[i][j][k] = 0;

data[0][0][0] = 1;
data[0][0][99] = 1;
data[0][99][0] = 1;
data[0][99][99] = 1;
data[99][0][0] = 1;
data[99][0][99] = 1;
data[99][99][0] = 1;
data[99][99][99] = 1;
```

Ausgehend von 4 Byte für den Datentyp *float* benötigt der Vektor 100*100*100*4 = 4 Millionen Byte im Speicher.

Zusatzfrage: Wie viel Kilobyte, bzw. Megabyte sind das?

13. `double table[20][2];`

14. **double** temperature[20][30][40];

15. **int** function(**int** data []);

16. **void** function(**double** data[][5]);

Lösungen zu Kapitel 7

1. Der *Pre-Compiler*.

2. Der *Linker*.

3. Wird bei einem **#include**-Befehl eine Headerdatei in spitzen Klammern angegeben, so wird die Datei nur in den Standard-Verzeichnissen für Headerdateien gesucht. Wird die Headerdatei in Anführungszeichen angegeben, so wird die Datei zunächst im aktuellen Arbeitsverzeichnis gesucht, bevor dann wieder in den Standard-Verzeichnissen für Headerdateien gesucht wird.

4. **#define** ANZAHL 100

5. **#define** DEBUG

6. ```
   #ifdef TEST
   printf("Dies ist ein Test!\n");
   #endif
   ```

7. Über Headerdateien wird einem Programmstück die Syntax von anderen Programmstücken mitgeteilt. Anders ausgedrückt: In eine Headerdatei sollten Deklarationen ausgelagert werden, die auch für Programmstücke in anderen Dateien wichtig sind.

8. Über Header-Dateien. Die Funktionsdeklarationen werden in eine separate Header-Datei verschoben. Die Header-Datei wird von anderen Programmblöcken mit dem **#include**-Befehl eingebunden.

9. Indem die gesamte Headerdatei durch einen **#ifndef**-Befehl eingeklammert wird. Beispiel:

   ```
   #ifndef HEADER_FILE_USER_INPUT_INCLUDED
   #define HEADER_FILE_USER_INPUT_INCLUDED

   int getInt(char *text);

   #endif /* #ifndef HEADER_FILE_USER_INPUT_INCLUDED */
   ```

10. Der *Compiler* wird pro Quellcode-Datei einmal aufgerufen, also dreimal, während der *Linker* danach nur einmal zum Verbinden der Programmblöcke gestartet wird.

11. Im professionellen Umfeld sollte immer nach der *top-down*-Methode vorgegangen werden. Die *bottom-up*-Methode bietet sich nur an, wenn *mal eben schnell* etwas ausprobiert werden soll und das Programmstück nicht Teil einer professionellen Software wird.
   ```

12. Siehe Abschnitt 7.5.1.
13. Siehe Abschnitt 7.5.1.
14. Bis die erstellte Software beim Kunden deinstalliert wird.

Stichwortverzeichnis