Grundlagen der Programmierung

Vorlesungsskript an der Fachhochschule Osnabrück

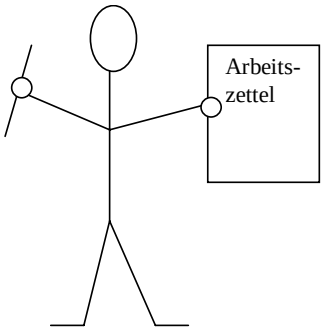
Prof. Dr. T. Gervens, Prof. Dr.-Ing. B. Lang, Prof. Dr.-Ing. C. Westerkamp,
Prof. Dr.-Ing. J. Wübbelmann

1 Aufbau und Arbeitsweise eines Rechners

1.1 Beispiel: Ein Rechenknecht

Aufgabe: Berechnung einer Quadratwurzel $\sqrt{x}$, $x > 0$ (näherungsweise).

Rechenknecht:



Möglichkeiten:
- Grundrechenarten (zählen)
- eine Zahl im Kopf merken
- positiv/negativ unterscheiden
- Absolutbetrag bilden

Hilfsmittel: Papier, Bleistift, Kopf, Finger

Unsere Aufgabe: Zerlegung in einzelne Rechenschritte, die der Rechenknecht ausführen kann.

Idee: Wähle Schätzwert s für $\sqrt{x}$ (z. B. 1)

Dann ergeben sich 3 Möglichkeiten:

i) $s = \sqrt{x}$ (Volltreffer, unwahrscheinlich)

ii) $s < \sqrt{x} \;\; (s > 0) \;\Rightarrow\; 1 < \frac{\sqrt{x}}{s} \;\Rightarrow\; \sqrt{x} < \frac{x}{s} \;\Rightarrow\; s < \sqrt{x} < \frac{x}{s}$

Der Zielwert liegt im Intervall zwischen *s* (untere Grenze) und $\frac{x}{s}$ (obere Grenze)

Nächster Schätzwert aus Mittelwert: $s' = \frac{s + \frac{x}{s}}{2}$;

Intervall verkleinert sich, bis die gewünschte Genauigkeit erreicht wird.

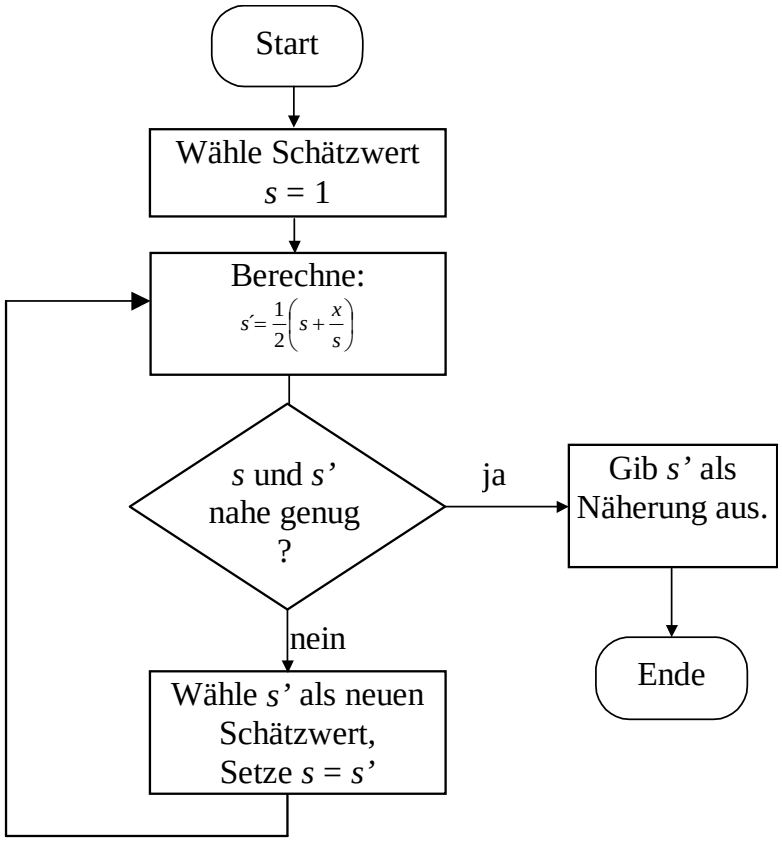
iii) $s > \sqrt{x} \;\Rightarrow\; 1 > \frac{\sqrt{x}}{s} \;\Rightarrow\; \sqrt{x} > \frac{x}{s} \;\Rightarrow\; \frac{x}{s} < \sqrt{x} < s$

Der Zielwert liegt im Intervall zwischen $\frac{x}{s}$ (untere Grenze) und *s* (obere Grenze)

Nächster Schätzwert aus Mittelwert: $s' = \frac{s + \frac{x}{s}}{2}$;

Intervall verkleinert sich, bis die gewünschte Genauigkeit erreicht wird.

Vorgehensweise (Flussdiagramm)



Rechenknecht bekommt ein Blatt mit entsprechenden Anweisungen und Daten. (1 Zeile = 1 Rechenanweisung oder 1 Datenfeld). Dabei gilt:

- **Alles** (Daten, Felder für Zwischenergebnisse, Anweisungen) soll auf dem Datenblatt dargestellt werden.
- **A** bezeichne die Zahl, die er sich im Kopf merkt.
- Bei **Operationen** ist ein Operand immer die Zahl „A“ im Kopf. Das Ergebnis steht wieder in A.

Arbeitszettel für Rechenknecht:

Zeilen-Nr	Inhalt	Kommentar
0	**Gehe zur Zeile 6**	
1	**1000 (Beispiel)**	Feld für x
2	**anfangs 1**	Feld für s
3	**anfangs undefiniert**	Feld für s’
4	**0,00001**	Konstante 0,00001
5	**2**	Konstante 2
6	**Inhalt von Zeile 1 nach A übertragen**	
7	**Teile A durch Zeile 2**	A=A/s
8	**Addiere Zeile 2 zu A**	A=A+s
9	**Teile A durch Zeile 5**	A=A/2
10	**Trage Inhalt von A in Zeile 3 ein**	s’
11	**Subtrahiere Zeile 2 von A**	A=s’-s
12	**Ersetze A durch seinen Absolutbetrag**	A=|A|
13	**Subtrahiere Zeile 4 von A**	A=A-0,00001
14	**Falls A≥0, gehe zu Zeile 17**	|s’-s|≥0,00001 ?
15	**Gebe den Inhalt von Zeile 3 aus**	Ergebnis: s’
16	**Halte an**	
17	**Inhalt von Zeile 3 nach A übertragen**	A=s’
18	**Inhalt von A nach Zeile 2 übertragen**	s=A (Ersetzung s=s’)
19	**Mache weiter mit Zeile 6**	

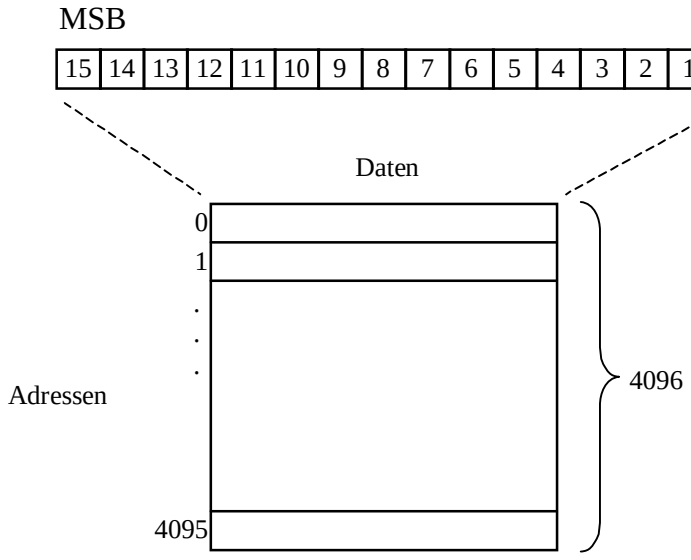
1.2 Der digitale Computer

Schrittweiser Übergang: Rechenknecht → DC (Digital Computer)

1. Arbeitszettel → Hauptspeicher

Der Hauptspeicher enthält ein Programm, dies umfasst sowohl Daten als auch Befehlsanweisungen.
Ein Programm stellt eine sinnvolle Folge von Befehlen mit zugehörigen Daten dar, die sequentiell abgearbeitet werden.

Beispiel: Speicher: - besteht aus 4K = 4096 = 2^{12} Worte
- jedes Speicherwort besteht aus 2 Bytes = 16 Bit
- Zugriff auf ein Wort unter Angabe der Adresse (direkter Zugriff/direkte Adressierung)



2. Gemerkte Zahl A → Akkumulator: 16 Bit Wort (ACC)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

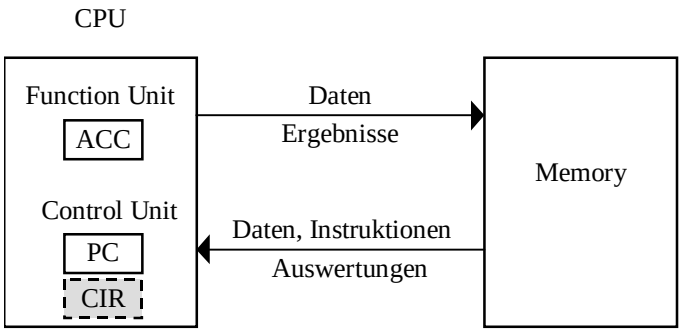
3. „Finger des Rechenknechts“ → Befehlszähler

(Kontrolle) PC (Program Counter)

Der Befehlszähler (PC) enthält immer die Adresse des nächsten auszuführenden Befehls.

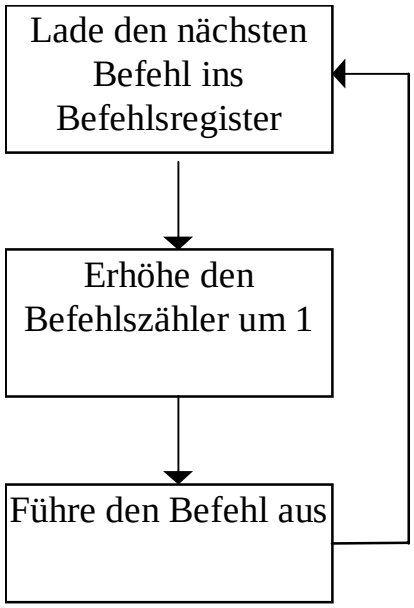
4. Kopf des Mannes → CPU (Central Processing Unit)

bestehend aus:
- Steuereinheit (Control Unit);
- Funktionseinheit (Function Unit, ALU)
- Befehlsregister (CIR: Current Instruction Register)



Die Steuereinheit entschlüsselt (dekodiert) Befehle und steuert deren Ausführung. Der aktuell bearbeitete Befehl befindet sich im Befehlsregister. Zu jedem Zeitpunkt führt die CPU genau einen Befehl aus.
Die Aufgabe der Funktionseinheit besteht in der klassischen Verarbeitung der Daten, d.h. Ausführungen von Berechnungen, sie wird von der Steuereinheit gesteuert.

5. Arbeitsweise: Fetch and Execute Cycle



6. Erweiterungen:

a) Programm Status Wort:

Merk-Bits, welche Information über die letzte Operation in der Funktionseinheit geben:

Carry Flag	Meldet Überläufe bei der Berechnung.
Parity Bit	Gibt an, ob eine gerade Anzahl von Einsen im Akku steht.
Sign Flag	Speichert das Vorzeichen der Zahl im Akku.
Zero Flag	Meldet, dass der Wert im Akku gleich Null ist.

b) Ein- und Ausgabegeräte (input und output devices)

- externe Speicher (Bänder, Platte, Disketten, etc.);
- Drucker
- Bildschirm
- Tastatur

c) BUS (Verbindung aller Elemente)

- Steuerleitungen;
- Datenbus (n Leitungen = n Bits werden in einem Takt übertragen);
- Adressbus (n Leitungen = 2^n Adressierungsmöglichkeiten)

d) Unterbrechungswerk

Zusatz: Unterbrechung wegen bestimmter Signale/Ereignisse ermöglichen

Ereignisse: -

- äußere (z. B. Reset)
- innere (z. B. Fertigmeldung eine E/A-Gerätes)
- Ausnahmen (z. B. ungültiger Befehlscode, Division durch 0 etc.)

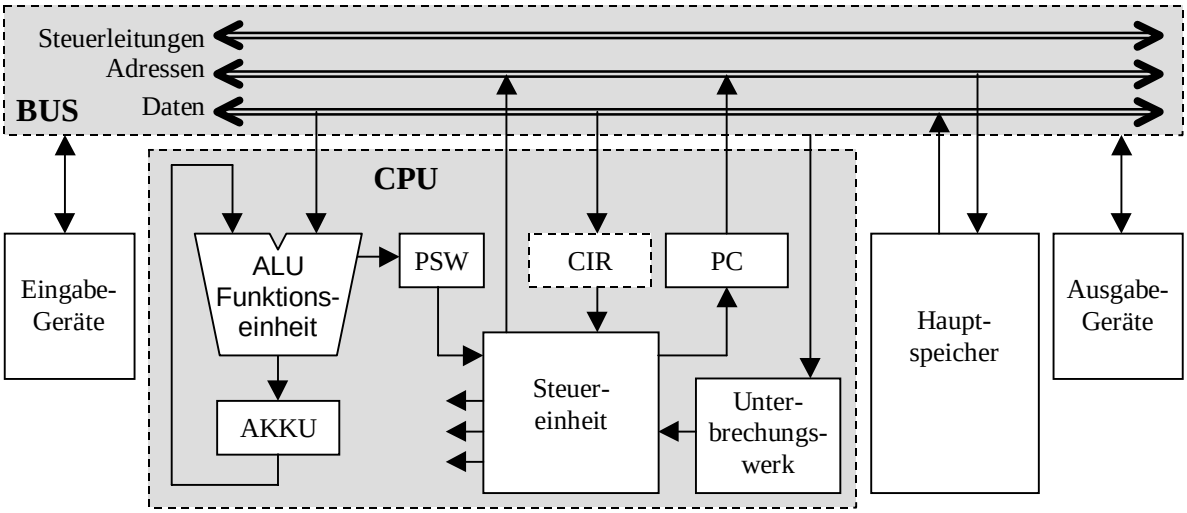
Reaktion des Prozessors

- unterbricht seinen Zyklus
- nimmt eine von Signalen abhängige Maßnahme vor
- setzt seinen Zyklus fort (wenn möglich)

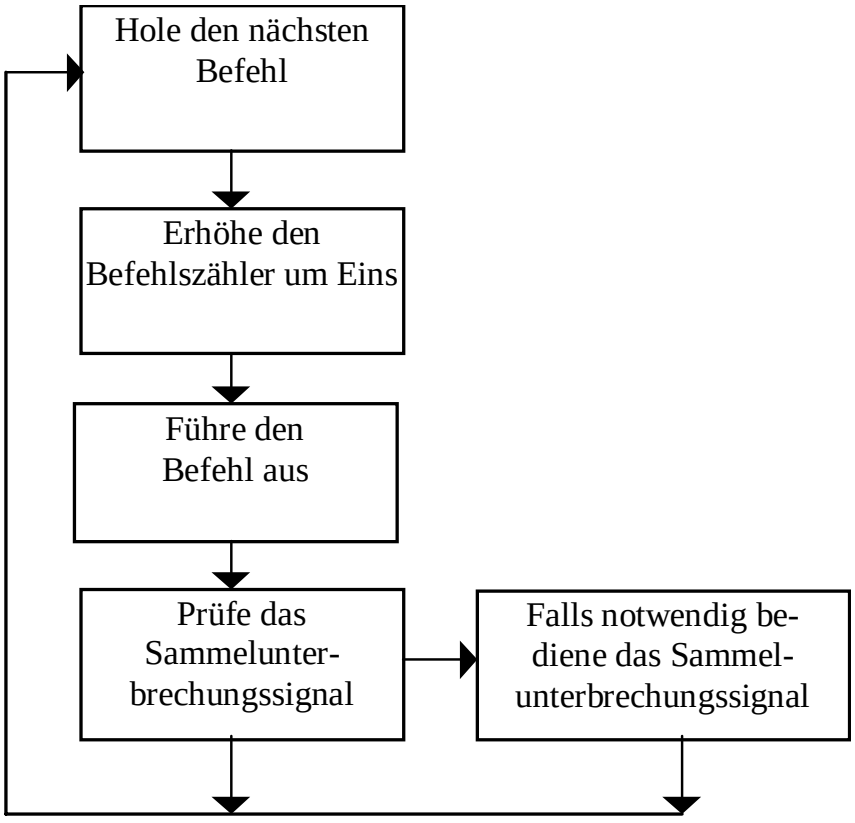
Dieser Zusatz wird vom **Unterbrechungswerk** verwaltet. Die Aufgaben des Unterbrechungswerkes sind dabei:

- Bildung des Sammelunterbrechungssignals
- Identifizierung der Startadresse einer Befehlssequenz im Hauptspeicher, die bei dieser Unterbrechung auszuführen ist
- Zurückweisen von Unterbrechungen
- Bestimmung der Rangfolge der Unterbrechungssignale

Grobstruktur eines von Neumann-Rechners

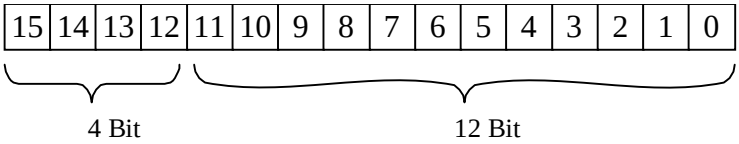


Der erweiterte Arbeitszyklus der CPU



1.3 Befehle des DC

Befehlsformat:



2^4 mögliche Befehle (16)
2^{12} mögliche Operandenadressen (4096)

Beispiel:

0010	100011001100	**Maschinensprache**
ADD	2252	**Assemblersprache, mnemonische Darstellung**

(Mnemonika: griech.: Gedächnis)

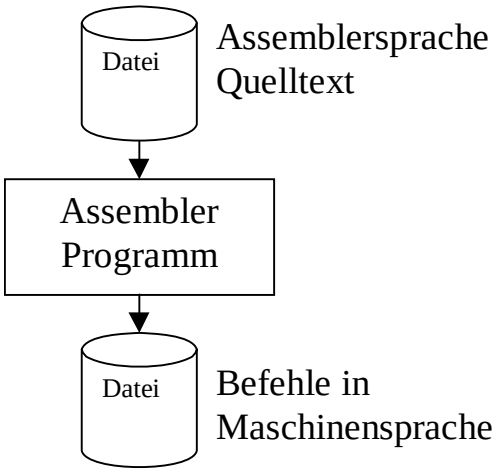
Mikroprogramm:

Für jeden Maschinenbefehl wird in der Steuereinheit ein Mikroprogramm ausgeführt, welches die einzelnen Schritte zur Ausführung des Befehls (Lesen Adresserzeugung, Ausführen, Unterbrechungsbehandlung) steuert. Das Mikroprogramm enthält Anweisungen, welche die Hardware direkt ansprechen.

Assemblerstruktur:

Mnemocode Operand

Der Assembler ist ein Programm, welches eine Datei, die Befehle in Assemblersprache (mnemonische Darstellung, Mnemocode) enthält, in Maschinensprache übersetzt.



Der Mnemocode wird 1:1 in Maschinencode übersetzt.
Symbolische Adressen (Befehle, Daten) werden aufgelöst und Absolutadressen zugeordnet.
Für Konstanten wird Code erzeugt.

DC - Befehle:
Der Beispielrechner DC soll die folgenden Befehle kennen.

1.3.1 Laden/Speichern

LDA X	Lade den Inhalt der Adresse X in den Akku
STA X	Speicher Akkuinhalt an der Adresse X

1.3.2 Arithmetische Befehle

ADD X	Addiere den Inhalt der Adresse X zum Akkuinhalt
SUB X	Subtrahiere den Inhalt der Adresse X vom Akku
MPY X	Multipliziere den Inhalt der Adresse X mit dem Akku
DIV X	Dividiere den Akkuinhalt durch die Adresse X

1.3.3 Logische Befehle

AND X	Bitweises UND des Akku mit der Adresse X
OR X	Bitweises ODER des Akku mit der Adresse X

1.3.4 Sprungbefehle

JMP X	Setze Verarbeitung mit der Anweisung bei X fort
JLE X	Sprung zur Adresse X, falls Akkuinhalt ≤ 0
JEQ X	Sprung zur Adresse X, falls Akkuinhalt = 0
JOV X	Sprung zur Adresse X, wenn ein „Overflow“ auftritt

1.3.5 Shift-Operator

SHR	zirkuläres Verschieben der Bits um eine Stelle nach „rechts“, Rotation, Bit ganz rechts wandert an die erste Stelle (ganz links)

1.3.6 Ein-/Ausgabebefehle

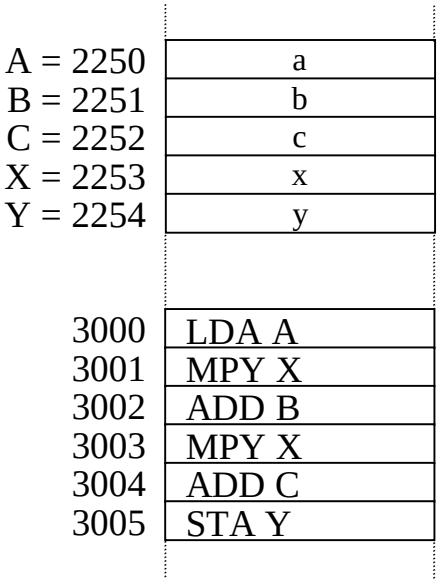
WTA id	Akkuinhalt auf das Gerät id ausgeben
RDA id	Wert des Gerätes id in den Akku einlesen
SKP id	Überspringen der nächsten Anweisung, falls Gerät id nicht „busy“

z. B.:
```
SKP id
JMP a
RDA id
```

Beispiele: Kurze DC-Programme

a) Berechnung von: $y = ax^2+bx+c = (ax+b)x+c$.

Start des Programmes bei Adresse 3000.



b) Maximum-Bestimmung von a und b

Speicherzelle mit symbolischer Adresse A enthält den Wert a = 3.
Speicherzelle mit symbolischer Adresse B enthält den Wert b = 5.
Speicherzelle mit symbolischer Adresse Max nimmt das Ergebnis auf.

Start des Programmes bei Zelle 100

Adresse	**Mnemocode**	**Akku-Inhalt**		**Befehlszähler(PC)**
100	LDA A	a	3	→ 101
101	SUB B	a-b	3-5=-2	→ 102
102	JLE 105	a-b	-2	→ 105
103	LDA A	a		→ 104
104	JMP 106	a		→ 106
105	LDA B	b	5	→ 106
106	STA Max	b	5	→ 107
107	... z.B. WTA 1			

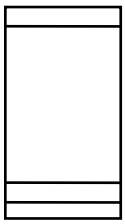
c) Sprung, wenn Akku > 0: JGT a

```
100   JLE   102
101   JMP   a
102   nächster Befehl...
```

1.4 Mikroprozessor 8080 (Vorläufer der 80x86- und Pentium-Prozessoren)

- Adressbus: 16 Bit = 2^{16} Adressen = 64 k, 1 Speicherwort: 8 Bit
- Datenbus: 8 Leitungen
- Steuerbus: 4 Leitungen

- Speicher:



- Befehlsablauf: sequentiell (Fetch and Execute)
  Abweichungen: - Pausen;
  - Programmsprünge;
  - Unterprogramme.

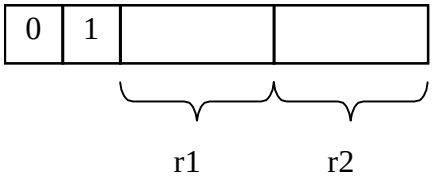
- Befehlsregister: 8 Bit plus 2 Hilfsregister für 1-3 Byte-Befehle

- 16 Bit Stapelanzeiger (Stackpointer SP)

- Programmstatusregister F (Flags)
  S: Sign-Flag
  Z: Zero-Flag
  P: Parity-Flag
  C: Carry-Flag
  AC: Auxiliary Carry Flag (Überlauf beim ersten Halbbyte beim Befehl DAA)

- Befehle (244):
  - 96 Befehle für Datenverschiebung
  - 65 Befehle für Arithmetik
  - 43 logische Befehle
  - 36 Befehle für Programmsprünge
  - 4 sonstige Steuerbefehle

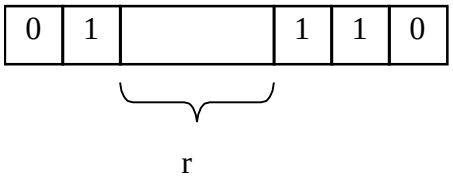
z. B. **MOV r1,r2** Inhalt Reg r2 → Reg r1



111 - Reg A 001 - Reg C
000 - Reg B 010 - Reg D

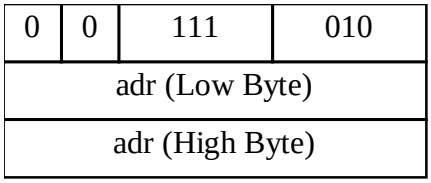
(3 Bits = 1 Register)

MOV r, M Inhalt Memory M → Reg r



M: H, L Register
(Die Speicheradresse wird durch H, L angegeben)

LDA adr Inhalt der Adresse adr in den Akku laden



2 Betriebssysteme

2.1 Überblick

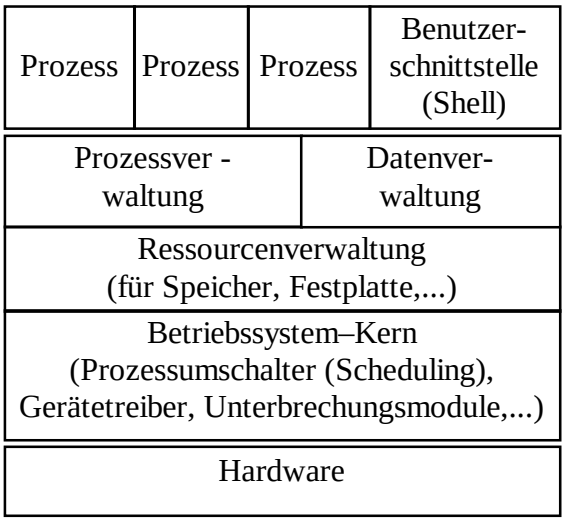
Das Betriebssystem stellt ein Interface (Schnittstelle) zwischen dem Benutzer und der Rechnerhardware dar. Es besteht aus einer Vielzahl von Programmen

- zur **Überwachung und Verwaltung der Hardware** (CPU, Hauptspeicher, Sekundärspeicher, Peripheriegeräte, Drucker, Terminals, etc.);
- zur **Steuerung des Prozessablaufs**, inklusive Prozesserzeugung, Prozessverdrängung und Prozesskommunikation;
- zur **Behandlung von Hard- und Softwarefehlern**, zu internen Diagnoseabläufen, zum Datenschutz und zur Datensicherheit:
- zur **Programmentwicklung** (Sprachübersetzer, Compiler, Editieren, Debugger, etc).

Bekannte Betriebssysteme:

- UNIX (Varianten: HP-UX, AIX, , XENIX, LINUX, SOLARIS, POSIX)
- MS-DOS
- VMS (Fa. DEC)
- OS/2
- Windows9x
- Windows NT; Windows 2000, Windows XP, Windows Vista
- Mac OS
- Echtzeitbetriebssysteme: VxWorks, OSE, pSOS, Windows Embedded, LINUX, CMX-RTX
- Betriebssysteme für kleine und mobile Plattformen: EPOC, Windows CE

Das Betriebssystem besteht aus mehreren Schichten:



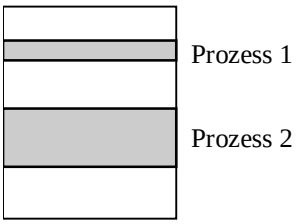
Die unteren Schichten sprechen die Hardware an (Gerätetreiber), bedienen Unterbrechungen (Unterbrechungsmodule) und ermöglichen das Umschalten (Scheduling) zwischen Prozessen. Weiterhin werden die Ressourcen des Rechners verwaltet. Diese sind z.B. der Hauptspeicher

und der Festplattenspeicher. Weitere Module des Betriebssystems verwalten die laufenden Prozesse und die Daten des Rechners (Dateisystem).
Die oberste Schicht ist die Prozessschicht. Hier laufen die Anwenderprozesse. Ein hervorgehobener Prozess ist die Benutzerschnittstelle. Diese ermöglicht das Bedienen des Rechners und das Starten von neuen Prozessen.
Unter UNIX erfüllt die Shell diese Aufgabe. Sie ermöglicht dem Benutzer die Eingabe von Kommandos als Text. Die Shell beinhaltet einen Kommandointerpreter, der die Kommandos auswertet und zu deren Ausführung zugehörige Prozesse startet. Alternativ kann eine graphische Oberfläche als Benutzerschnittstelle dienen. Unter Windows dient dazu der Explorer, der in verschiedenen Ausprägungen (Desktop, Windows-Explorer, Internet-Explorer) dem Benutzer die Bedienung des Rechners ermöglicht. Auch bei UNIX kann eine grafische Bedienung des Rechners erfolgen. Basierend auf der Grafiksoftware X-Windows und Aufsätzen wie KDE sind Dateimanager-Programme ähnlich dem Windows-Explorer verfügbar, welche ein interaktives Bedienen des Rechners ermöglichen.
Bei eingebetteten Geräten (Steuerungen, Telekommunikationsgeräten) geht der Trend zu browser-basierten Benutzerschnittstellen (Web-Server im Gerät).

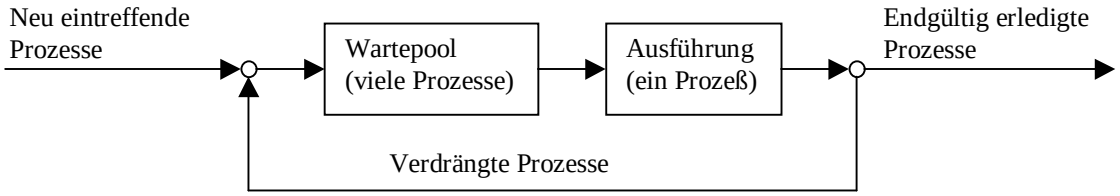
2.2 Prozessverwaltung

2.2.1 Scheduling

Ein Prozess (Job, Task) ist die Instanz eines ausführbaren Programmes im Speicher, die ausgeführt wird, oder zur Ausführung ansteht. In einem System mit einem Prozessor kann immer genau ein Prozess ausgeführt werden.

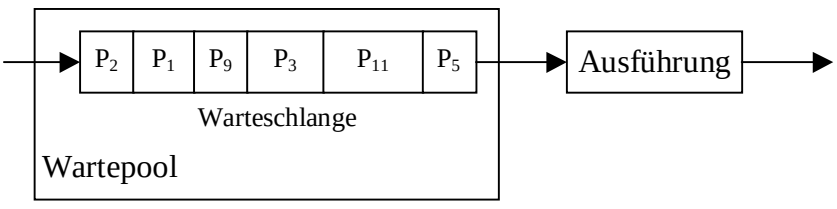


Das Betriebssystem hat die Aufgabe des **Scheduling's**: Es wählt den Prozess aus, der zur Ausführung kommt und bestimmt wann ein Prozess verdrängt wird.



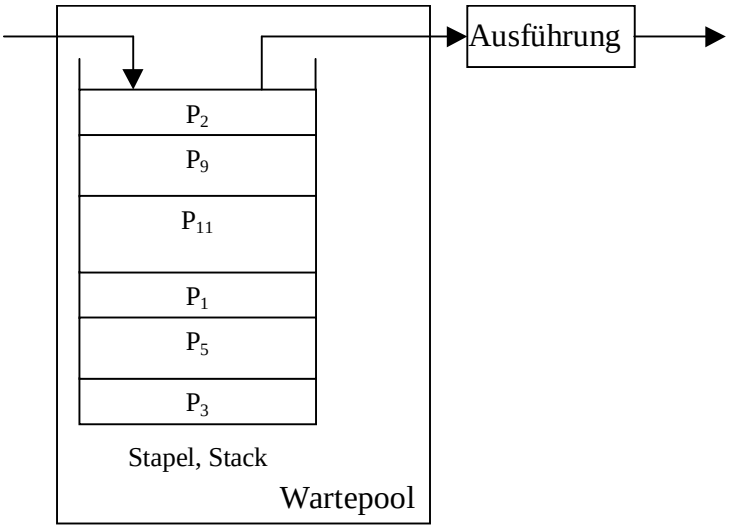
Scheduling Strategien

1. FIFO (First In First Out)



Bei Verwendung eines Wartepools mit FIFO-Strategie werden die Prozesse in der Reihenfolge, mit der sie in den Wartepool eingestellt wurden, zur Ausführung gebracht.

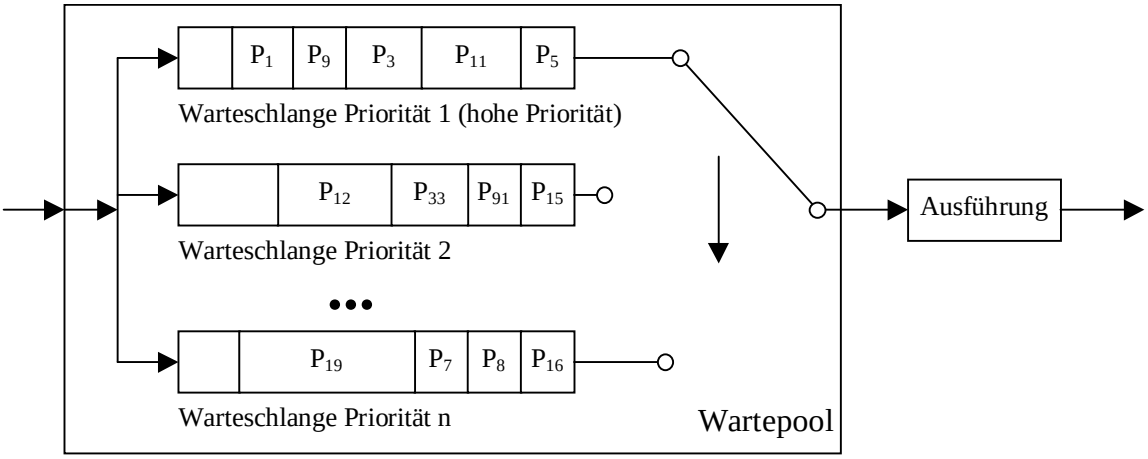
2. LIFO (Last In First Out)



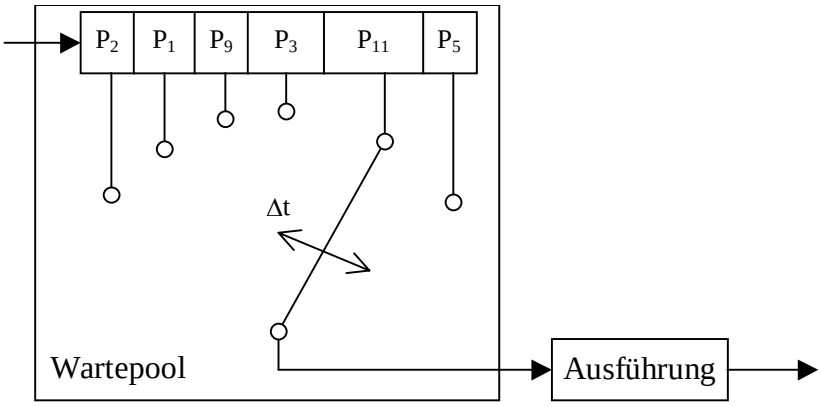
Bei Verwendung eines Wartepools mit LIFO-Strategie (Stapel, Stack) wird der Prozess, der zuletzt in der Wartepool eingestellt wurde, als erster zur Ausführung gebracht.

3. Prioritätensteuerung

Jeder Benutzer erhält eine bestimmte Priorität und für jede Priorität gibt es eine Warteschlange. Prozesse mit hoher Priorität werden vorgezogen.



4. Time Sharing–System



Das Time Sharing System bringt die Prozesse nach einem bestimmten Zyklus für einen Zeitabschnitt Δt (Zeitscheibe) zur Ausführung, dann schaltet es zu einem anderen Prozess um. Damit wird ein Prozess schrittweise ausgeführt bis er abgearbeitet ist.

Im praktischen Einsatz werden meist die Scheduling-Strategien kombiniert. In UNIX findet man beispielsweise eine Kombination des Time Sharing mit der Prioritätenvergabe.

Andere weitergehende Scheduling-Strategien (inkl. Optimierungsstrategien) werden benötigt im:

- Mehrprozessorbetrieb
- Parallel Processing
- Massive Parallel Processing

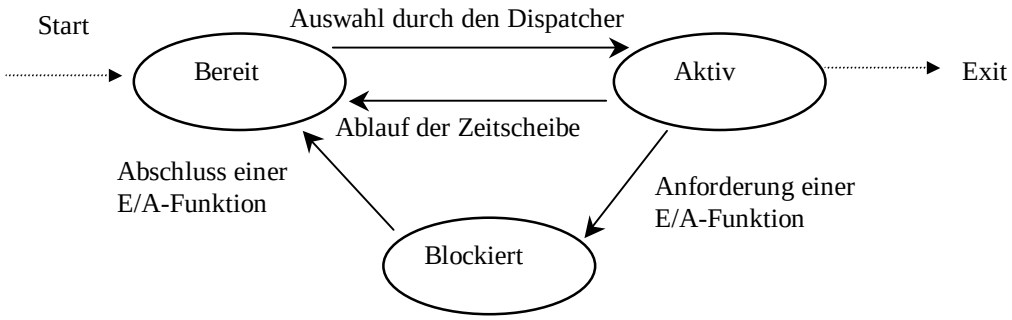
2.2.2 Mögliche Prozesszustände

1. Prozess ist aktiv (Unterschiede: Benutzermodus/Systemmodus).
2. Prozess ist nicht aktiv, aber bereit, d. h. wartet auf Zuweisung durch den Scheduler.
3. Prozess ist blockiert, d. h. wartet auf eine E/A-Operation.

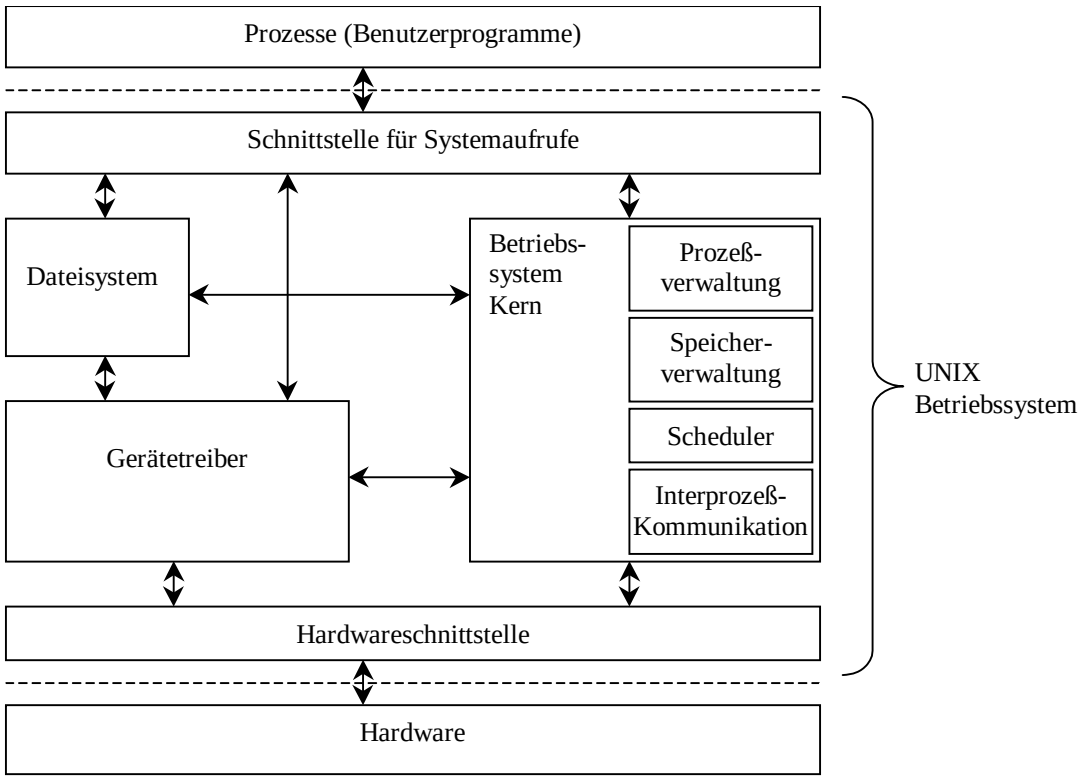
Für die Umschaltung auf einen neuen Prozess ist der Dispatcher zuständig. Er wird angestoßen, falls:

- das Zeitquantum für einen Prozess abgelaufen ist (Time Sharing)
- ein aktiver Prozess eine E/A-Anforderung stellt
- ein Prozess beendet ist

Nachfolgendes Diagramm zeigt ein einfaches Modell der Prozesszustände und beschreibt die Übergänge zwischen den Zuständen für ein Time Sharing System:

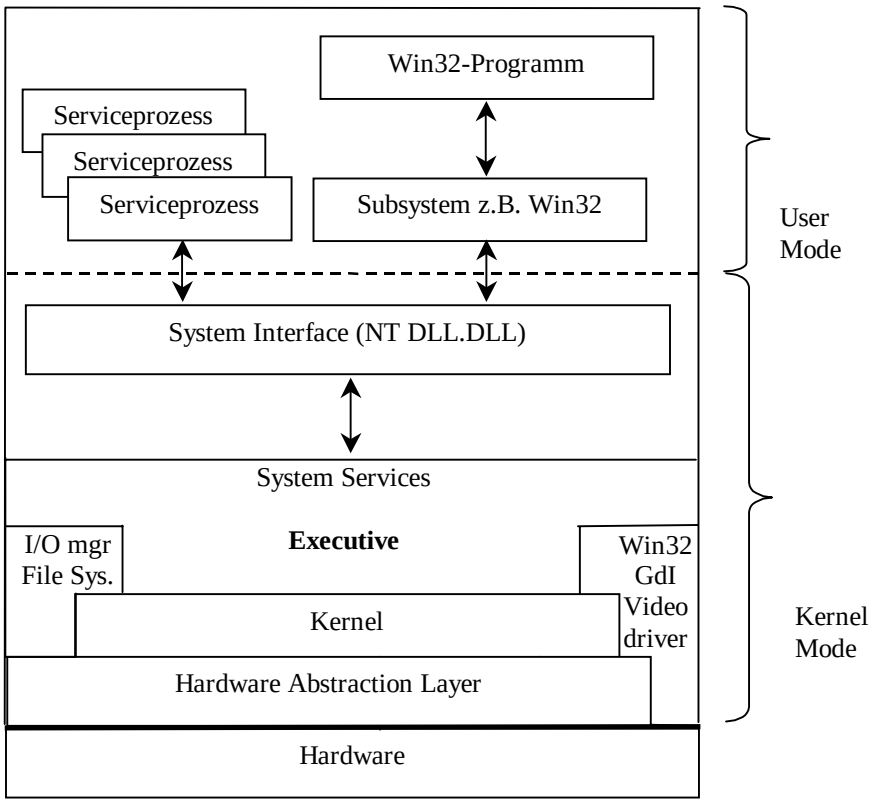


2.2.3 UNIX



Das UNIX Betriebssystem erlaubt das Ausführen mehrerer Prozesse nach Prioritätensteuerung und Time Sharing. Die Prozesse kommunizieren mit dem Betriebssystem über Systemaufrufe, für welche eine eindeutige Schnittstelle existiert.
Das Betriebssystem UNIX ist unterteilt in mehrere Komponenten. Der Betriebssystem Kern verwaltet den Speicher und die laufenden Prozesse. Der Scheduler ist für das Umschalten zwischen den Prozessen zuständig. Weiterhin ermöglicht der Kern eine Kommunikation zwischen Prozessen (Interprozesskommunikation).
Die Gerätetreiber kapseln Hardwareeinheiten und geben unterschiedlicher Hardware die gleiche Software-Schnittstelle. Beispielsweise werden unterschiedliche Festplatten oder Grafikkarten durch Treiber gekapselt und können einheitlich durch die Software angesprochen werden.
Das Dateisystem organisiert die Plattenspeicher (Festplatte, Diskette, CD-ROM ...) und ermöglicht den Zugriff über eine logische Dateistruktur. Diese Dateistruktur bildet das Dateisystem ab auf die physikalischen Spuren und Sektoren der Platten.
UNIX ist unabhängig vom Prozessor (Pentium. PowerPC, StrongARM, ...). Ein kleiner Teil wird an den Prozessor angepasst. Dieser Teil ist in einer Hardwareschnittstelle zusammengefasst.
UNIX ist hauptsächlich in der Sprache C programmiert. Teile der Hardwareschnittstelle müssen in der Assemblersprache des Prozessors geschrieben werden.

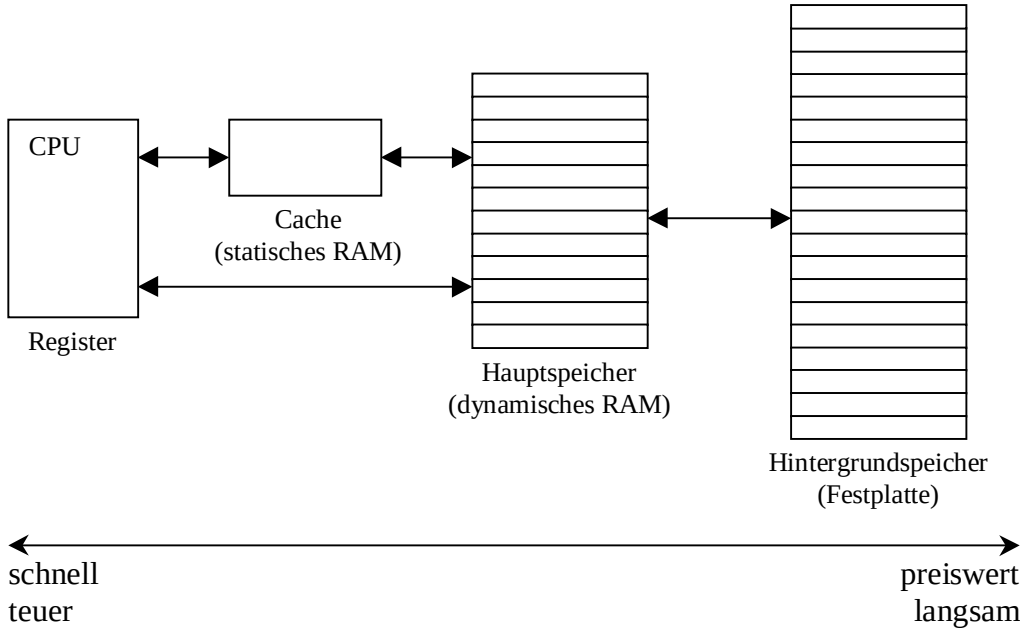
2.2.4 Windows



Windows entstand als grafische Benutzeroberfläche für das einfache PC-Betriebssystem DOS. Dieses war für genau einen Prozessortyp (X86) in großen Teilen in Assembler programmiert. Seit Windows NT ist gibt es Versionen für mehrere Prozessorplattformen, auch wenn weiterhin der Schwerpunkt auf Pentium-Prozessoren (z.B. Windows XP) bzw. StrongARM (Windows CE, Windows Mobile) liegt.
Die unterliegende Hardware wird (bis auf direkten Video-Zugriff) durch einen Hardware Abstraction Layer (HAL) gekapselt. Dieser überträgt alle Hardware-Zugriffe des Kernels und des Dateisystems an die Hardware. Der Executive enthält mehrere Management-Dienste, die Steuerung von Objekten, Cache, Prozessen, Konfiguration etc. übernehmen. Alle Operationen unterhalb der System-Services-Ebene finden im Kernel-Mode statt und sind damit dem direkten Zugriff von Benutzerprogrammen entzogen. Bei der Erzeugung von Kernel-Objekten (Dateien, Prozesse, Threads. Pipes etc.) erhält der aufrufende Prozess ein „Handle“, über das die weitere Nutzung des Objektes geschieht. Benutzerprogramme nutzen die sehr umfangreiche Win32-API (Application Programming Interface). So gibt es über 60 Funktionen für Dateizugriff und mehr als 1000 für die grafische Benutzeroberfläche. Viele Funktionen lassen sich auf mehrere Arten realisieren. Der Zugriff geschieht über das Win32-Subsystem auf die System Services. Weitere Programmiermodelle und Subsysteme sind in einigen Versionen realisiert (z.B. OS/2 bis Windows NT). Die Verwaltung der Konfiguration und der Eigenschaften von Benutzerprogrammen geschieht über eine große Datenbank (Registry).

2.3 Speicherverwaltung

2.3.1 Speicherhierarchie



In Rechnern besteht eine Speicherhierarchie mit den Registern in der CPU an der Spitze und dem Hintergrundspeicher an der Basis. Der Speicher an der Spitze der Hierarchie ist sehr schnell im Zugriff aber auch sehr teuer. Der Hintergrundspeicher an der Basis ist preiswert, aber der Zugriff auf diesen Speicher ist langsam.

Cache-Speicher stellt ein Bindeglied zwischen CPU und Hauptspeicher dar, in dem die als nächstes benötigten Daten zwischengespeichert sind. (90:10-Regel: Bei 90% aller Zugriffe werden nur 10% der Daten verwendet). Speichert man diese 10% der Daten in einem schnellen Cache-Speicher, können somit 90% aller Zugriffe beschleunigt durchgeführt werden.

Für die Speicherverwaltung wird zwischen physikalischem und logischem Adressraum unterschieden:

- physikalischer Adressraum: reale Hauptspeicher
- logischer Adressraum: (virtuell) der durch ein Programm ansprechbare Adressraum.

Die Umsetzung zwischen logischen und physikalischen Adressen erfolgt mit Hardware und wird durch das Betriebssystem gesteuert. Der Programmierer von Benutzerprogrammen braucht sich darum nicht kümmern.

2.3.2 Auslagern von Programmen

Bei gleichzeitiger Ausführung mehrerer Prozesse reicht der Hauptspeicher als Speicherraum nicht aus. Daher werden dann Teile der Programme in den Hintergrundspeicher ausgelagert. Das Betriebssystem führt nach bestimmten Strategien (Paging, Swapping) dieses Auslagern durch.
Hierzu werden Hauptspeicher und Hintergrundspeicher in Seiten (Pages) bzw. Kacheln (Page-Frames) gleicher Größe (z. B. 4K) unterteilt. Die im Programm angesprochene Adresse steht nun entweder im Haupt- oder im Hintergrundspeicher. Steht sie nicht im Hauptspeicher (Page-Fault), so wird sie aus dem Hintergrundspeicher nachgeladen und

für die Verarbeitung verfügbar gemacht. Ist im Hauptspeicher kein Platz, so muss eine andere Seite des Hauptspeichers in den Hintergrundspeicher ausgelagert werden.

Auslagerungsstrategien

- FIFO (First In First Out)
  Die am längsten im Hauptspeicher liegende Seite wird ausgelagert.
- LRU (Least Recently Used)
  Die am längsten nicht benutzte Seite wird ausgelagert.
- LFU (Least Frequently Used)
  Die bislang am wenigsten benutzte Seite wird ausgelagert.

Bei jeder Strategie sind entsprechende Listen von Zugriffen oder Zeiten zu führen.

In der Praxis: LRU

Einlagerungsstrategien

- demand paging Die Seite wird bei aktuellem Bedarf angefordert.
- preplanned paging Der Seitentransport wird vorgeplant, so dass die Seite bei Bedarf schon zur Verfügung steht. Hierzu sind Kenntnisse über den Programmablauf erforderlich.

Übliche Strategie: demand paging

2.4 Dateiverwaltung

Die Dateiverwaltung gibt dem Anwender eine logische Sicht auf die Daten der Massenspeicher (Festplatten, Disketten, ...) des Rechners. Diese logische Sicht basiert auf Dateien und Verzeichnissen.

Datei: Einheit zusammenhängender Zeichen.
In der Regel werden diese Zeichen dem Programm in sequentieller Reihenfolge zur Verfügung gestellt.
Eine Datei besitzt einen Dateinamen und ist einem Verzeichnis zugeordnet.

Verzeichnis: Zugriffspfad auf Dateien innerhalb einer Verzeichnisstruktur.
Verzeichnisstrukturen sind üblicherweise in einer Baumstruktur angelegt.

Die Dateiverwaltung muss somit die physikalische Struktur der Massenspeicher umsetzen auf die gewünschte logische Sicht. Zu berücksichtigen sind:

- die physikalische Sicht (Zugriff, Struktur, Aufbau, Ablage, Einteilung in Blöcke)
- die logische Sicht (Dateiname, Dateiverzeichnisse, Dateiorganisation, Dateiformate, Datensicherheit, Datenschutz)

Dateiname: In vielen Betriebssystemen besteht der Name aus zwei Feldern, dem Namensfeld und dem Typfeld (DOS, VMS, OS/2, nicht UNIX)

Bsp.:	Schach.c	(C-Programm)
	Schach.obj	(übersetztes Programm)
	Schach.exe	(ausführbares Programm)
	Readme.txt	(zu lesender Text)
	DV.doc	(z. B. Word Dokument)

In UNIX sind Dateinamen nicht strukturiert. Der Punkt kann Bestandteil des Namens sein. Steht der Punkt an erster Stelle, so wird die Datei beim Auflisten der Dateien nicht mit angezeigt.

Durch Verwendung von Kürzelzeichen (Wildcard) können mehrere Dateien gleichzeitig angesprochen werden.

z.B. UNIX:
- `*` null oder mehrere Zeichen
- `?` genau ein beliebiges Zeichen (kein Leerzeichen!)
- `[..]` genau eins der Zeichen in der Klammer, wobei Abkürzungen wie a-z oder 1-20 möglich sind.

Bsp.:		
	text.*	text.txt, text.1, text.doc, ...
	tex?.txt	text.txt, tex1.txt, tex2.txt, ...
	text.[1-4]	text.1, text.2, text.3, text.4
	a.[a-c]	a.a, a.b, a.c

Verzeichnisse: Dateien werden in der Regel in hierarchisch angelegten Verzeichnissen (Directories) angeordnet. (In UNIX: ein einziger Baum für alle Datenträger.)

/	root directory
/bin	Dienstprogramme
/etc	weitere Programme
/etc/passwd	Passwords
/usr	
/home5/g65/std7372	

Der volle Dateiname besteht aus Pfadname und Dateiname, z. B.:
/home6/westerka/dv.doc

2.5 Liste einiger UNIX-Befehle

login <username>	Anmelden
who	Anzeige der aktiven Benutzer
date	Datum und Uhrzeit
pwd	aktuelles Verzeichnis wird angezeigt
cd /	gehe ins Wurzelverzeichnis
cd	gehe ins Home-Verzeichnis
cd ..	gehe ins übergeordnete Verzeichnis
cd <pfadname>	gehe ins Verzeichnis <pfadname>
mkdir	Verzeichnis erstellen
rmdir	Verzeichnis löschen
ls	Listing der Dateien
ls -l	Listing mit ausführlichen Informationen
ls -a	Listing inkl. versteckter Dateien
rm	Löschen von Dateien
cp	Kopieren
mv	Verschieben/Umbenennen von Dateien
cat	Anzeigen von Dateien
more	seitenweises Anzeigen von Textdateien
chmod	Ändern der Zugriffsrechte (ugo±rwx)
gedit *&*	Aufruf des Texteditors als Hintergrundprozess
xemacs &	Aufruf des Editors XEmacs
ps	Programm zur Anzeige der Prozesse
passwd	Ändern des Paßwortes
cc oder gcc	Aufruf des C-Compilers
grep muster datei.*	Finden muster in datei.*
mp test.c >test.ps	Erstellen einer Postscript-Datei
ftp	Programm zur Datenübertragung auf Basis des Internetprotokolls TCP/IP
lpr -Pa4ps	Ausdrucken einer Postscript -Datei auf dem Schnelldrucker

2.6 Programmiersprachen

Programmiersprachen sind künstliche Sprachen zur präzisen Formulierung von Algorithmen. Ein Algorithmus ist dabei ein allgemeiner Lösungsweg der festlegt, wie man von der Aufgabenstellung schrittweise zur Lösung kommt. Ein Programm ist die Codierung eines Algorithmus in einer Programmiersprache.

2.6.1 Klassen von Programmiersprachen

Man unterscheidet folgende Klassen von Programmiersprachen:

Sprachen der 1. Generation: Maschinensprachen

Sprachen der 2. Generation: Assemblersprachen

Sprachen der 3. Generation:

FORTRAN (ab 1954, wissenschaftliche und technische Anwendungen)
COBOL (ab 1957, kommerzielle Anwendungen)
ALGOL (1957, wissenschaftliche und technische Berechnungen)
PL/1
BASIC (1963, einfache Problemstellungen, Lehre)
Pascal (1968, allgemeine Probleme, Lehre)
C (1970, allgemeine Probleme, Systemimplementierungsaufgaben)
Ada (1975, Systemimpl. vor allem im sicherheitskritischen Bereichen)
Modula (1975, Lehre)
C++ (1980, objektorientierte Anwendungsprogr., Benutzeroberflächen)
JAVA (1995, objektorientierte Sprache, portabel, netzwerkfähig)

Sprachen der 4. Generation: Toolsprachen, Datenbanksprachen wie SQL

Sprachen der 5. Generation: deklarative Sprachen zur Beschreibung des Problems (Forschungsgegenstand).

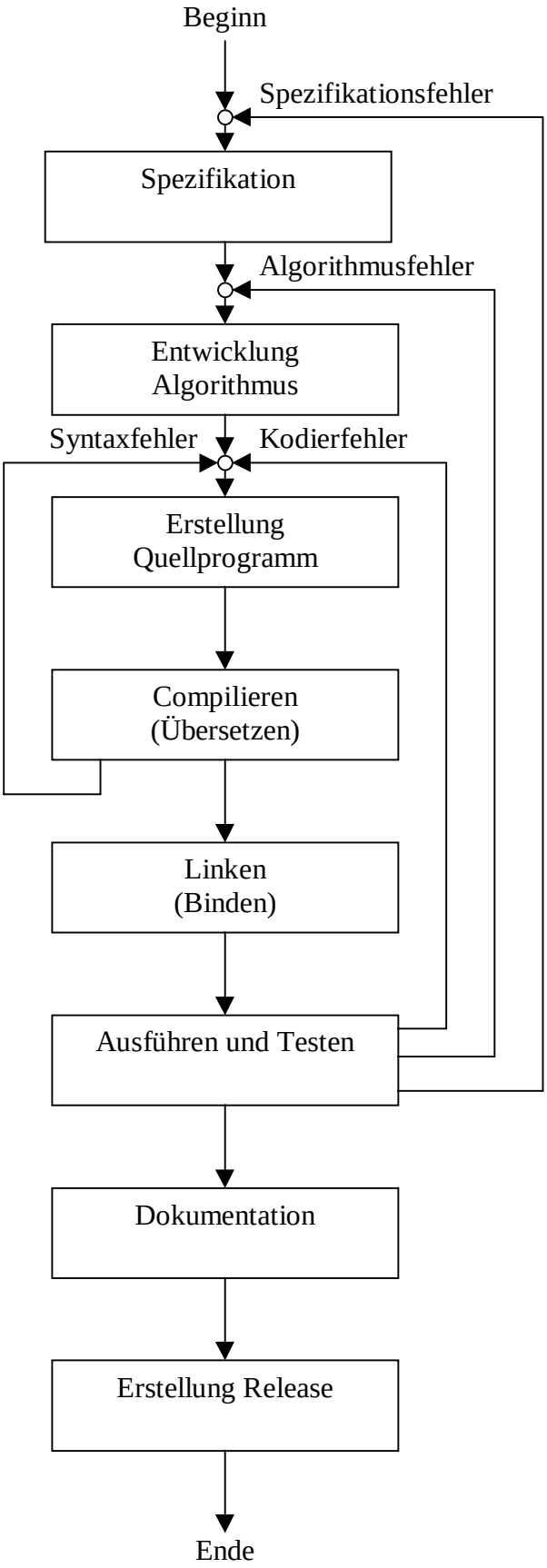
In der Vorlesung und in den Praktika beschäftigen wir uns kurz mit Assemblersprachen (2. Generation) und ausführlich mit C (3. Generation).

2.6.2 Erstellung eines Programms

Die Programmierung einer Aufgabenstellung erfolgt in mehreren Schritten

1. **Spezifikation** der Aufgabenstellung: Textuelle und graphische Darstellung des Problems, welches durch das Programm gelöst werden soll. Eine möglichst genaue Spezifikation des Verhaltens, welches das Programm zeigen soll, verhindert Missverständnisse zwischen dem Auftraggeber (Kunden) und dem Auftragnehmer (Programmierer). Die Spezifikation sollte vom Auftraggeber und vom Auftragnehmer abgenommen werden.
2. Erstellen und Skizzieren eines **Lösungsplans (Algorithmus).** Ein solcher Lösungsplan kann als **Verlaufsplot, Struktogramm** oder in **Pseudocode** beschrieben werden**.**
3. Erstellen des sogenannten **Quell-Programms** (Source-Code) in der vorgegebenen Sprache mit Hilfe eines ASCII-Editors (bei uns mit dem Texteditor oder Emacs).
4. **Übersetzen/Compilieren** des Quellprogramms in Maschinensprache (Objektprogramm). Dieser Schritt wird durch ein Programm ausgeführt, welches als „Compiler“ bezeichnet wird. Der im Praktikum verwendete Compiler heißt „cc“ oder „gcc“. Beim Compilieren führt das Compiler-Programm mehrere Schritte durch:
    - In einer *lexikalischen Analyse* werden logisch-zusammenhängende Tokens gelesen.
    - In einer *syntaktischen Analyse* wird die Struktur des Quellprogramms aufgrund von Regeln erkannt.
    - Schließlich wird der *Maschinencode* erzeugt und anschließend *optimiert*.

5. **Binden (Linken):**Das Objektprogramm (evtl. auch mehrere) wird durch Hilfsprogramme (bei uns mit cc oder gcc)zu einer ausführbaren Datei gebunden bzw. gelinkt. Die ausführbare Datei heißt bei UNIX und Verwendung des cc- oder gcc-Compilers standardmäßig „a.out“.
6. **Ausführen und Testen:** Das Programm wird in den Hauptspeicher des Rechners geladen und ausgeführt. Es wird getestet, ob Spezifikation erfüllt wird, sonst muss nachgebessert werden (Zurück zu Punkt 3, Punkt 2 oder sogar zu Punkt 1, je nach Fehler).
   Zum Testen von Programmen dienen häufig „Debugger“-Programme, die ein „Hineinsehen“ in den Programmcode und die Inspektion von Variablen während des Programmlaufs ermöglichen.
7. **Dokumentation:** Nach erfolgreichem Test des Programms muss es dokumentiert werden. Die Dokumentation soll es einem fremden Programmierer ermöglichen, in kurzer Zeit den Quellcode und die hinterliegenden Algorithmen zu verstehen, um das Programm warten und modifizieren zu können.
8. **Erstellung einer Release:** In einer Release stellt man alle Informationen (Spezifikation, Lösungsplan, Quellcode, Anweisungen zum Übersetzen des Quellcodes, verwendete Compiler, Dokumentation) eines Programms zusammen, um zu einem späteren Zeitpunkt aus dem Release wieder das Programm erstellen zu können.
   Hilfe bei der Release-Erstellung bieten Konfigurationsmanagement-Tools (Versionsverwaltung). Sie speichern die Dateien eines aktuellen Entwicklungsstands mit Versionsinformation und ermöglichen jederzeit ein Abrufen genau dieses Standes.


Beginn
Spezifikationsfehler
Spezifikation
Algorithmusfehler
Entwicklung
Algorithmus
Syntaxfehler
Kodierfehler
Erstellung
Quellprogramm
Compilieren
(Übersetzen)
Linken
(Binden)
Ausführen und Testen
Dokumentation
Erstellung Release
Ende

Grundlagen der Programmierung

Vorlesungsskript der Fachhochschule Osnabrück

Prof. Dr. T. Gervens, Prof. Dr.-Ing. B. Lang, Prof. Dr.-Ing. C. Westerkamp,
Prof. Dr.-Ing. J. Wübbelmann

3 Informationsdarstellung

3.1 Zeichendarstellung

Digitalanlagen (*digitus* lat. Finger) arbeiten mit elektrischen Signalen, durch welche die Daten dargestellt werden. Die Darstellung muss mit Grundelementen realisiert werden, für die es nur zwei Zustände gibt: 0 oder 1. Ein Grundelement, welches die Zustände 0 oder 1 realisieren kann heißt Bit (Binary digit).

Zeichentypen:

- Alphabet
- Ziffern
- Steuerzeichen
- Operationszeichen
- Sonderzeichen (§, #, ...)
- Grafikzeichen (□, |, ...)
- Satzzeichen (!, ?, ;, ...)

Notwendig ist eine Codierung der Zeichen durch eine Bitfolge, d.h. ein Bitmuster.

Mit	einem	Bit lassen sich	zwei	Zeichen darstellen.
Mit	zwei	Bit lassen sich	vier	Zeichen darstellen.
Mit	drei	Bit lassen sich	acht	Zeichen darstellen.
…				
Mit	n	Bit lassen sich	2^n	Zeichen darstellen.

Bezeichnungen:

4Bit	⇔	1 Nibble	Unterscheidung von 16 Zeichen
8 Bit	⇔	1 Byte	Unterscheidung von 256 Zeichen
16/32/64 Bit	⇔	1 Wort	(Anzahl ist Rechnerabhängig)

Speichergrößen

1KibiByte = 1024 Byte
1 MebiByte = 1024 KibiByte
1 GibiByte = 1024 MebiByte

Historisch wird ein KiByte oft noch als KByte bezeichnet, ein MiByte als MByte, usw.

Die Norm IEC 60027-2 (seit 2000) definiert folgende SI-Binäreinheiten:

Binäreinheit: Name, Symbol		Dezimaleinheit: Name, Symbol		Abweichung:
Kibi, Ki	2^{10}	Kilo, k	10^3	2,40%
Mebi, Mi	2^{20}	Mega, M	10^6	4,86%
Gibi, Gi	2^{30}	Giga, G	10^9	7,37%
Tebi, Ti	2^{40}	Terra, T	10^{12}	9,95%
Pebi, Pi	2^{50}	Peta, P	10^{15}	12,60%
Exbi, Ei	2^{60}	Exa, E	10^{18}	15,29%
Zebi, Zi	2^{70}	Zetta, Z	10^{21}	18,06%

Yobi, Yi	2^{80}	Yotta, Y	10^{24}	20,89%

Konvention zur Zeichendarstellung:

ASCII (American Standard Code for Information Interchange)
ursprünglich 7 Bit ⇔ 128 Zeichendarstellung

Beispiel:

U = 1010101
W = 1010111
E = 1000101

Der ASCII-Code berücksichtigt Kleinbuchstaben, Großbuchstaben und die Ziffern von 0 bis 9 in alphabetischer bzw. natürlicher Reihenfolge. Sprachspezifische Sonderzeichen wie die deutschen Umlaute kommen jedoch nicht vor. In Europa wird häufig die 8-Bit-ASCII-Erweiterung Latin-1 verwendet, die durch die Norm ISO8859-1 beschrieben wird. Doch UNIX-Rechner sowie die meisten Programmiersprachen benutzen nur den genormten ASCII-Zeichensatz von 0 bis 127.

Die neuere Programmiersprache Java baut auf den neueren Zeichensatz **Unicode** auf, der praktisch alle weltweit geläufigen Zeichen bis hin zu japanischen oder tibetischen Schriftzeichen umfasst. Der Unicode ist eine 16 Bit Codierung und kennt somit maximal $65636=2^{16}$ Zeichen. Die untersten 128 Zeichen des Unicode stimmen mit den Zeichen des ASCII-Codes überein.

Tabelle 1: ASCII-Code mit hexadezimaler Bezeichnung der Codeworte

00	NUL	10	DLE	20	SP	30	0	40	@	50	P	60	`	70	p
01	SOH	11	DC1	21	!	31	1	41	A	51	Q	61	a	71	q
02	STX	12	DC2	22	„	32	2	42	B	52	R	62	b	72	r
03	ETX	13	DC3	23	#	33	3	43	C	53	S	63	c	73	s
04	EOT	14	DC4	24	$	34	4	44	D	54	T	64	d	74	t
05	ENQ	15	NAK	25	%	35	5	45	E	55	U	65	e	75	u
06	ACK	16	SYN	26	&	36	6	46	F	56	V	66	f	76	v
07	BEL	17	ETB	27	,	37	7	47	G	57	W	67	g	77	w
08	BS	18	CAN	28	(	38	8	48	H	58	X	68	h	78	x
09	HT	19	EM	29	)	39	9	49	I	59	Y	69	i	79	y
0A	LF	1A	SUB	2A	*	3A	:	4A	J	5A	Z	6A	j	7A	z
0B	VT	1B	ESC	2B	+	3B	;	4B	K	5B	[	6B	k	7B	{
0C	FF	1C	FS	2C	´	3C	<	4C	L	5C	\	6C	l	7C	|
0D	CR	1D	GS	2D	-	3D	=	4D	M	5D	]	6D	m	7D	}
0E	SO	1E	RS	2E	.	3E	>	4E	N	5E	^	6E	n	7E	~
0F	SI	1F	US	2F	/	3F	?	4F	O	5F	_	6F	o	7F	DEL

Tabelle 2: Bezeichnungen der Steuerzeichen des ASCII-Codes

NUL	Null	FF	Form feed	CAN	Cancel
SOH	Start of heading	CR	Carriage return	EM	End of medium
STX	Start of text	SO	Shift out	SUB	Substitute
ETX	End of text	SI	Shift in	ESC	Escape
EOT	End of transmission	DLE	Data link escape	FS	File separator
ENQ	Enquiry	DC1	Device control 1	GS	Group separator
ACK	Acknowledge	DC2	Device control 2	RS	Record separator
BEL	Bell	DC3	Device control 3	US	Unit separator
BS	Backspace	DC4	Device control 4	SP	Space
HT	Horizontal tab	NAK	Negative acknowledge	DEL	Delete
LF	Line feed	SYN	Synchronous idle		
VT	Vertical tab	ETB	End of transmission block		

3.2 Zahlendarstellung

3.2.1 Ganze Zahlen ohne Vorzeichen

Dezimalsystem: $(59310)_{10} = 5\cdot10^4 + 9\cdot10^3 + 3\cdot10^2 + 1\cdot10^1 + 0\cdot10^0$

Dualsystem: $(1011)_2 = 1\cdot2^3 + 0\cdot2^2 + 1\cdot2^1 + 1\cdot2^0$

Allgemein: Die Darstellung im Stellenwertsystem zur Basis B ($B\in N, B>1$)

$$(a_{n-1}a_{n-2}a_{n-3}...a_0)_B = a_{n-1}B^{n-1} + a_{n-2}B^{n-2} + a_{n-3}B^{n-3} + ... + a_0B^0 = \sum_{i=0}^{n-1} a_iB^i$$

wobei a_i n verschiedene Zeichen aus der Menge {0 ... B-1} sind.

Beispiele:

a) B=10 Ziffern: 0 ... 9
$(3728)_{10} = 3\cdot10^3 + 7\cdot10^2 + 2\cdot10^1 + 8\cdot10^0$

b) B=2 Ziffern 0, 1
$(10110)_2 = 1\cdot2^4 + 0\cdot2^3 + 1\cdot2^2 + 1\cdot2^1 + 0\cdot2 = 22$

c) B=16 Ziffern: 0 ... 9, A ... F
$(A0)_{16} = A\cdot16^1 + 0\cdot16^0 = 160$

d) B=3 Ziffern: 0, 1, 2
$(2120)_3 = 2\cdot3^3 + 1\cdot3^2 + 2\cdot3^1 + 0\cdot3^0 = 69$

3.2.1.1 Umrechnung: B-Stellenwertsystem → Dezimalsystem

(mit Hilfe des Horner Schemas)

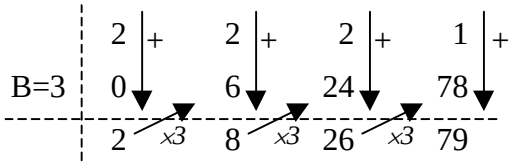
Beispiel:

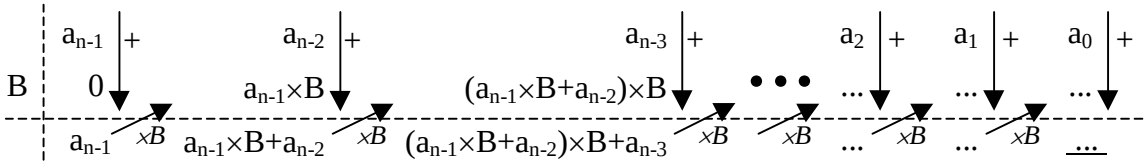
$(2221)_3 = 2\cdot3^3 + 2\cdot3^2 + 2\cdot3^1 + 1\cdot3^0$
$= ((2\cdot3 + 2)\cdot3 + 2)\cdot3 + 1\cdot3^0 = (8\cdot3 + 2)\cdot3 + 1 = 79$

Allgemeines Schema

$$(a_{n-1}a_{n-2}a_{n-3}...a_1a_0)_B = ((...((a_{n-1}B + a_{n-2})B + a_{n-3})B + ... + a_2)B + a_1)B + a_0$$

Methode (Horner)





Beispiele:

a) $(111001)_2$

	1	1	1	0	0	1
2		2	6	14	28	56
	1	3	7	14	28	57

b) $(11036)_7$

	1	1	0	3	6
7		7	56	392	2765
	1	8	56	395	2771

c) $(1234)_5$

	1	2	3	4
5		5	35	190
	1	7	38	194

d) $(ABCDEF)_{16}$

	A	B	C	D	E	F
16		160	2736	43968	703696	11259360
	10	171	2748	43981	703710	11259375

e) $(2CA)_{16}$

	2	C	A
16		32	704
	2	44	714

3.2.1.2 Umrechnung Dezimalsystem → B-Stellenwertsystem

Die Zahl wird nacheinander durch B dividiert. Die entstehenden Reste bilden die Ziffern im B-Stellenwertsystem, angefangen mit der 0-ten Stelle.

$$\left(a_{n-1}a_{n-2}a_{n-3}\ldots a_2a_1a_0\right)_B = a_{n-1}B^{n-1} + a_{n-2}B^{n-2} + \ldots + a_2B^2 + a_1B^1 + a_0$$

1. Teilung durch B liefert: $x_1 = a_{n-1}B^{n-2} + a_{n-2}B^{n-3} + \ldots + a_2B + a_1$ Rest: a_0
2. Teilung durch B liefert: $x_2 = a_{n-1}B^{n-3} + a_{n-2}B^{n-4} + \ldots + a_2$ Rest: a_1

…

n-te Teilung durch B liefert: $x_n=0$ Rest: a_{n-1}

Beispiel:

a) $(101)_{10}=(?)_3$

101:3= 33 R: 2 = a_0
33:3= 11 R: 0 = a_1
11:3= 3 R: 2 = a_2
3:3= 1 R: 0 = a_3
1:3= 0 R: 1 = a_4

Ergebnis: $(101)_{10}=(10202)_3$

b) $(200)_{10}=(?)_2$

200:2= 100 R: 0
100:2= 50 R: 0
50:2= 25 R: 0
25:2= 12 R: 1
12:2= 6 R: 0
6:2= 3 R: 0
3:2= 1 R: 1
1:2= 0 R: 1

Ergebnis: $(200)_{10}=(11001000)_2$

c) $(200)_{10}=(?)_{16}$

200:16 = 12 R: 8
12:16 = 0 R: 12 $(12)_{10}=(C)_{16}$
Ergebnis: $(200)_{10}=\underline{(C8)}_{16}$

3.2.1.3 Umrechnung Dualsystem ↔ Hexadezimalsystem

Tabelle 3: Gegenüberstellung von Zahlenkodierungen

Dezimal		Dual		Hexadezimal	
0		0		0	
1	10^0	1	2^0	1	16^0
2		10	2^1	2	
3		11		3	
4		100	2^2	4	
5		101		5	
6		110		6	
7		111		7	
8		1000	2^3	8	
9		1001		9	
10	10^1	1010		A	
11		1011		B	
12		1100		C	
13		1101		D	
14		1110		E	
15		1111		F	
16		1 0000	2^4	10	16^1
17		1 0001		11	
18		1 0010		12	
19		1 0011		13	
20		1 0100		14	
21		1 0101		15	
22		1 0110		16	
23		1 0111		17	
24		1 1000		18	
25		1 1001		19	
26		1 1010		1A	
27		1 1011		1B	
28		1 1100		1C	
29		1 1101		1D	
30		1 1110		1E	
31		1 1111		1F	
32		10 0000	2^5	20	
...		...		...	

Vorgehen bei der Umrechnung von Dualzahlen nach Hexadezimalzahlen:

a) Einteilung des Bitmusters in Vierergruppen (von rechts).
b) Man ermittle die zu jeder Vierergruppe äquivalente Hexadezimalziffer.
c) Man ersetze jede Vierergruppe durch die zugehörige Hexadezimalziffer.

Beispiel: Umrechnung einer 8stelligen Dualzahl:

$(a_7\ a_6\ a_5\ a_4\ a_3\ a_2\ a_1\ a_0)_2$

$= a_7\,2^7 + a_6\,2^6 + a_5\,2^5 + a_4\,2^4 + a_3\,2^3 + a_2\,2^2 + a_1\,2^1 + a_0 2^0$

$= \underbrace{(a_7\,2^3 + a_6\,2^2 + a_5\,2^1 + a_4\,2^0)}_{h_1}\,2^4 \;+\; \underbrace{(a_3\,2^3 + a_2\,2^2 + a_1\,2^1 + a_0 2^0)}_{h_0}$

$= (h_1\ h_0)_{16}$

Analog lassen sich bei Dualzahlen mit höherer Stellenanzahl Vierergruppen bilden, die gruppenweise als Ziffer ins Hexadezimalsystem überführt werden.

Für die Umrechnung Hexadezimalzahlen nach Dualzahlen führe man die analogen Schritte in umgekehrter Reihenfolge aus:

a) Zerlegung der Hexadezimalzahl in ihre Hexadezimalziffern.
b) Ermittlung der zur Hexadezimalziffer äquivalenten dualen Vierergruppe.
c) Man ersetze jede Hexadezimalziffer durch die zugehörige Vierergruppe im Dualsystem.

Beispiele:

a) $(10\ 1000\ 0111\ 1001)_2 = (\ 2\ \ 8\ \ 7\ \ 9\ )_{16}$
b) $(\ 7\ \ A\ \ 1\ \ F\ )_{16} = (0111\ 1010\ 0001\ 1111)_2$
c) $(110\ 1010\ 0111\ 0001\ 1110\ 1111)_2 = (\ 6\ \ A\ \ 7\ \ 1\ \ E\ \ F\ )_{16}$

3.2.2 Darstellung positiver und negativer ganzer Zahlen (Zweierkomplement)

In den gängigen Rechnern werden Wortlängen von 8 -, 16 -, 32 - oder 64 Bit verwendet.
In einem 16 Bit = 2 Byte Wort lassen sich beispielsweise positive Zahlen im Bereich von 0 bis 2^{16}-1 darstellen (0 bis 65535):

f_{15}	f_{14}	f_{13}	f_{12}	f_{11}	f_{10}	f_9	f_8	f_7	f_6	f_5	f_4	f_3	f_2	f_1	f_0

2. Byte 1. Byte

Die Darstellung der negativen Zahlen durch Vorzeichen (1 Bit) und Absolutwert (Betrag) hätte zwei Nachteile:

- für Null gibt es zwei Darstellungen
- Rechenwerk muss subtrahieren können

Deshalb: Die Darstellung negativer Zahlen erfolgt durch das sogenannte Zweierkomplement!
Das Zweierkomplement zu x (Länge n Bits) ist $\tilde{x} = 2n - x$
Dann ist $\tilde{x} + x = 2^n$. Der Wert 2^n entspricht 0, da in einem Wort der Länge n die Zahl 2^n nicht darstellbar ist.
Im Rechner wird für negative Zahlen statt -x der 2er-Komplementwert $\tilde{x}$ von x verwendet.

Darstellung positiver und negativer Zahlen:

positive Zahlen:	$0 \le x \le 2^{n-1}-1$	→	normale duale Darstellung $f_{n-1} = 0$, x positiv
negative Zahlen:	$-2^{n-1} \le x < 0$	→	statt x wird $\tilde{x} = 2^n - \lvert x \rvert$ dargestellt $\tilde{x} \ge 2^n - 2^{n-1} = 2^{n-1}$, d.h. $f_{n-1} = 1$

Beispiele:

a) Stelle 3 und -3 mit 4 Bit (Länge n = 4) im Dualsystem mit Hilfe des Zweierkomplements dar.

f_3	f_2	f_1	f_0

$x = 3 = (0011)_2$. Darstellung :

0	0	1	1

$x = -3,\ \tilde{x} = 2^n - |x| = 2^4 - 3 = 16 - 3 = 13 = (1101)_2$ Darstellung:

1	1	0	1

Probe:

	0	0	1	1	≅	3
	1	1	0	1	≅	-3
1	0	0	0	0	≅	0

b) Stelle 5 und -5 mit 4 Bit (Länge n = 4) im Dualsystem mit Hilfe des Zweierkomplements dar.

$x = 5 = (0101)_2$. Darstellung :

0	1	0	1

$x = -5,\ \tilde{x} = 2^n - |x| = 2^4 - 5 = 16 - 5 = 11 = (1011)_2$. Darstellung:

1	0	1	1

Probe:

	0	1	0	1	≅	5
	1	0	1	1	≅	-5
1	0	0	0	0	≅	0

Bestimmung des 2er-Komplements

Das 2er-Komplement läßt sich auch ohne Subtraktion in zwei Schritten ermitteln:

1) Bestimmung des 1er-Komplements
   $x \rightarrow (2^n-1) - x$ Umkippen (Invertieren) aller Bits
2) Addition von 1
   $\tilde{x} = ((2^n-1) - x) + 1 = 2^n - x$

Beispiele:

a) Darstellung von -5 mit Wortlänge n = 4
   1. Schritt: $x = 5 = (0101)_2 \rightarrow (1010)_2$
   2. Schritt:

```
  1010
+    1
------
  1011
```

   $(1011)_2$ = 2er-Komplement von x = 5

b) Darstellung von -4 mit Wortlänge n = 4
   1. Schritt: $x = 4 = (0100)_2 \rightarrow (1011)_2$
   2. Schritt:

```
  1011
+    1
------
  1100
```

   $(1100)_2$ = 2er-Komplement von x = 4

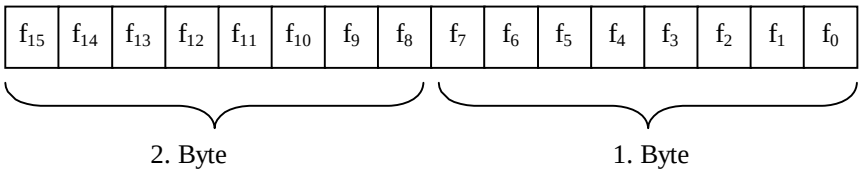
Darstellung aller 2er-Komplementzahlen mit n = 4

0	0000		
1	0001	-1	1111
2	0010	-2	1110
3	0011	-3	1101
4	0100	-4	1100
5	0101	-5	1011
6	0110	-6	1010
7	0111	-7	1001
		-8	1000

3.2.3 Byte-Reihenfolge (Endianess)

In Rechnersystemen sind die kleinsten adressierbaren Einheiten im allgemeinen Bytes. Eine Adresse bestimmt damit ein Byte im Speicher, die nächste Adresse das nächste Byte.

Ein 16 Bit Wort belegt beispielsweise 2 Byte die einzeln adressiert werden können.



In diesem Beispiel enthält das 1. Byte die Bits f_0 - f_7 und wird als niederwertiges Byte (*Least Significant Byte*, LSB) bezeichnet. Die Bits f_8 - f_{15} sind im 2. Byte enthalten, dieses wird als höchstwertiges Byte (*Most Significant Byte*, MSB) bezeichnet.

Werden nun Datenstrukturen mit mehr als einem Byte im Speicher angelegt, ergeben sich mehrer Möglichkeiten:

- Das Byte mit den höchstwertigen Bits auf der niedrigeren Adresse gespeichert, die Bytes mit den niederwertigen Bits an den folgenden Adressen im Speicher. In diesem Fall wird vom **Big-Endian** Mode gesprochen.
- Das Byte mit den niederwertigen Bits wird auf der niedrigeren Adresse gespeichert, die Bytes mit den höherwertigen Bits an den folgenden Adressen im Speicher. In diesem Fall wird vom **Little-Endian** Mode gesprochen.

Beide Modi finden in aktuellen Prozessorfamilien Anwendung. So wird der Big-Endian Mode u.a. von SPARC, PowerPC Prozessoren unterstützt. Little-Endian Prozessoren sind u.a. die x86 Familie (Pentium).
Wichtig ist die Byte-Reihenfolge bei Netzwerkprotokollen. Das Internet Protokoll verwendet z.B. die Big-Endian Reihenfolge. Die Daten dürfen damit nicht ohne weiteres in den Sendepuffer von Little-Endian Maschinen geschrieben werden, sondern müssen vorher angepasst werden. Ansonsten ist die Byte-Reihenfolge für Hochsprachenprogrammierer weitgehend transparent, da der Compiler sich um die Adressierung kümmert.

Beispiel: Die vorzeichenlose 32 Bit Zahl
$(305419896)_{10} = (12345678)_{16} = (0001\ 0010\ 0011\ 0100\ 0101\ 0110\ 0111\ 1000)_2$
soll ab Adresse 1000 gespeichert werden.

	Big-Endian			Little-Endian		
Adresse	Hex	Dez	Binär	Hex	Dez	Binär
1000	12	18	0001 0010	78	120	0111 1000
1001	34	52	0011 0100	56	86	0101 0110
1002	56	86	0101 0110	34	52	0011 0100

1003	78	120	0111 1000	12	18	0001 0010

3.2.4 Zweierkomplementarithmetik

Zunächst: Addition zweier einziffriger Dualzahlen:

1. Summand	2. Summand	Summe	Übertrag
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Summe von 2 einziffrigen Dualzahlen + Übertrag (3 Summanden)

1. Summand	2. Summand	Übertrag (3. Summand)	Summe	Übertrag
0	0	0	0	0
1	0	0	1	0
0	1	0	1	0
1	1	0	0	1
0	0	1	1	0
1	0	1	0	1
0	1	1	0	1
1	1	1	1	1

Addition zweier Dualzahlen mit n Stellen:

Obige Ziffernarithmetik wird verwendet, um eine wortweise Addition s = a+b zweier Dualzahlen $a = (a_{n-1}\ a_{n-2}\ ...\ a_1\ a_0)_2$ und $b = (b_{n-1}\ b_{n-2}\ ...\ b_1\ b_0)_2$ zur Summe $s = (s_{n-1}\ s_{n-2}\ ...\ s_1\ s_0)_2$ zu realisieren. Mittels der Übertragsbits $(ü_n\ ü_{n-1}\ ...\ ü_2\ ü_1)$ werden die Überträge bei der Berechnung in die höheren Stellen weitergereicht:

$(ü_1\ s_0)_2 = a_0 + b_0$
$(ü_2\ s_1)_2 = a_1 + b_1 + ü_1$
$(ü_3\ s_2)_2 = a_2 + b_2 + ü_2$
:
$(ü_{n-1}\ s_{n-2})_2 = a_{n-2} + b_{n-2} + ü_{n-2}$
$(ü_n\ s_{n-1})_2 = a_{n-1} + b_{n-1} + ü_{n-1}$

Beispiel 1: (Wortlänge = 4)

$x_1 = 7 = (0111)_2$ $\quad \tilde{x}_1 = 1001$

$x_2 = 4 = (0100)_2$ $\quad \tilde{x}_2 = 1100$

a) $x_1 - x_2$

$x_1 =$	0111
$+ \tilde{x}_2 =$	1100
Überträge	1100
Summe	0011 $\cong 3$

b) $x_2 - x_1$

$x_2 =$	0100
$+ \tilde{x}_1 =$	1001
Überträge	0000
Summe	1101 $\cong -3$

Das Ergebnis ist negativ!

Beispiel 2: (Wortlänge = 8, Bereich:$[-2^7, 2^7-1]$)

$x_1 = 55 = (00110111)_2$ $\quad \tilde{x}_1 = 11001001$

$x_2 = 33 = (00100001)_2$ $\quad \tilde{x}_2 = 11011111$

a) $x_1 - x_2 = 55-33$

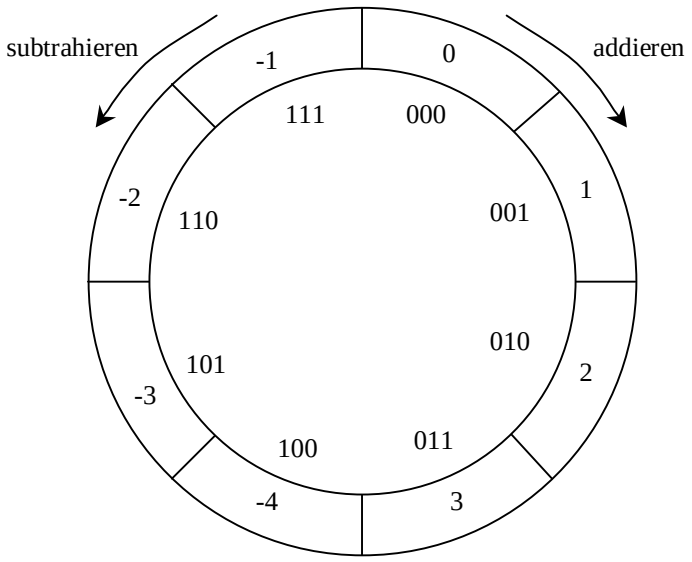
$x_1 =$	00110111
$+ \tilde{x}_2 =$	11011111
Überträge	11111111
Summe	00010110 22_{10}

b) $x_2 - x_1 = 33-55$

$x_2 =$	00100001
$+ \tilde{x}_1 =$	11001001
Überträge	00000001
Summe	11101010 -22_{10}

Rückkomplementieren von 11101010 ergibt: 00010110 22_{10}

Die arithmetischen Operationen inklusive der eventuellen Überläufe lassen sich im sogenannten Zahlenkreis verdeutlichen. Additionen sind Bewegungen im Uhrzeigersinn, Subtraktionen gegen den Uhrzeigersinn.



3.2.5 Darstellung gebrochener Zahlen

Umwandlung gebrochener Zahlen: Dezimal → B-Stellenwertsystem

Gebrochene Zahlen zu einer Basis B lassen sich wie folgt darstellen:

$$(a_{n-1}\, a_{n-2}\ \ldots\ a_1\, a_0, a_{-1}\, a_{-2}\ \ldots\ a_{-m})_B = a_{n-1}B^{n-1} + a_{n-2}B^{n-2} + \ \ldots\ + a_1 B^1 + a_0 + a_{-1}B^{-1} + a_{-2}B^{-2} + \ \ldots\ + a_{-m}B^{-m}.$$

Die gebrochenen Zahlen bestehen somit aus einem ganzzahligen Teil und einem Dezimalbruch (Nachkommateil).
Die Wandlung des ganzzahligen Teils wurde schon in Abschnitt 3.2.1.1 vorgestellt.
Der Dezimalbruch läßt sich durch schrittweises Ausklammern von B^{-1} wie folgt umformen:

$$(0, a_{-1}\, a_{-2}\, a_{-3} \ldots a_{-m+1}\, a_{-m})_B = B^{-1}(a_{-1} + B^{-1}(a_{-2} + B^{-1}(a_{-3} + \ \ldots\ + B^{-1}(a_{-m+1} + B^{-1}a_{-m}) \ldots)).$$

Die Ziffern $a_{-1}, \ldots, a_{-m}$ ergeben sich schrittweise durch Multiplikation des Dezimalbruchs mit der Basis B und Abspaltung des dabei entstehenden ganzen Teils. Dieser ganze Teil ergibt in jedem Schritt eine Ziffer des Dezimalbruchs.

1. Multiplikation → $a_{-1} + B^{-1}(a_{-2} + B^{-1}(a_{-3} + \ \ldots\ + B^{-1}(a_{-m+1} + B^{-1}a_{-m}) \ldots))$
2. Multiplikation → $a_{-2} + B^{-1}(a_{-3} + \ \ldots\ + B^{-1}(a_{-m+1} + B^{-1}a_{-m}) \ldots)$
...
m. Multiplikation → a_{-m}

Die Umwandlung gebrochener Zahlen aus dem Dezimalsystem in das B-Stellenwertsystem erfolgt somit in folgenden Schritten

1. Ganzzahligen Teil umwandeln gemäß Kap. 3.2.1.1.
2. Dezimalbruch umwandeln durch Schrittweise Multiplikation.
3. Ergebnis aus 1. und 2. zusammenfassen.

Beispiele:

a) $42{,}625 = (???)_2$

$42{,}625 = 42 + 0{,}625$

Umwandlung des ganzzahligen Teils		Umwandlung des Dezimalbruchs (Nachkommateil)			
42 : 2 =	21 R 0 → $a_0 = 0$	0,625	· 2 = 1,25	= 1 + 0,25	→ $a_{-1} = 1$
21 : 2 =	10 R 1 → $a_1 = 1$	0,25	· 2 = 0,5	= 0 + 0,5	→ $a_{-2} = 0$
10 : 2 =	5 R 0 → $a_2 = 0$	0,5	· 2 = 1,0	= 1 + 0	→ $a_{-3} = 1$
5 : 2 =	2 R 1 → $a_3 = 1$				
2 : 2 =	1 R 0 → $a_4 = 0$				
1 : 2 =	0 R 1 → $a_5 = 1$				

$42 = (101010)_2 \quad 0{,}625 = (0{,}101)_2 \ \Rightarrow\ 42{,}625 = (101010{,}101)_2$

b) $0{,}2 = (???)_2$

0,2	· 2 = 0,4	= 0+ 0,4	$a_{-1} = 0$
0,4	· 2 = 0,8	= 0 + 0,8	$a_{-2} = 0$
0,8	· 2 = 1,6	= 1 + 0,6	$a_{-3} = 1$
0,6	· 2 = 1,2	= 1 + 0,2	$a_{-4} = 1$
0,2	· 2 = 0,4	ab hier periodisch	

$0{,}2 = \left(0, \overline{0011}\right)_2$

c) $0{,}825 = (???)_{16}$

0,825	· 16 = 13,02	= 13 + 0,2	$a_{-1} = D$
0,2	· 16 = 3,02	= 3 + 0,2	$a_{-2} = 3$
0,2	· 16 = 3,02	= 3 + 0,2	$a_{-3} = 3$

$$0{,}825 = \left(0, D\bar{3}\right)_{16}$$

Verschiebung des Kommas

Beispiele: $123{,}45 = 0{,}12345\times10^{3} = 12345\times10^{-2}$
$(101{,}11)_2 = 0{,}1011\times2^{3} = 10111 \cdot 2^{-2}$

Allgemeiner kann eine Verschiebung des Kommas um 2 Stellen nach rechts wie folgt dargestellt werden:

$$\left(a_{n-1}a_{n-2}a_{n-3}...a_0, a_{-1}a_{-2}...a_{-m}\right)_B = \left(a_{n-1}a_{n-2}a_{n-3}...a_0a_{-1}a_{-2}, a_{-3}...a_{-m}\right)\times B^{-2}$$

Eine Verschiebung des Kommas um j Stellen kann durch Multiplikation mit B^j bzw. B^{-j} ausgeglichen werden:

Verschiebung nach links: Multiplikation mit B^j
Verschiebung nach rechts: Multiplikation mit B^{-j}

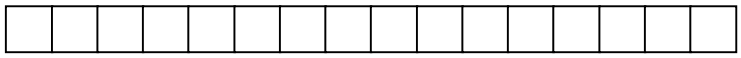
Umrechnung B-Stellenwertsystem → Dezimal

Verschiebung des Kommas, bis ein ganzzahliger Teil entsteht, dann umwandeln wie gewohnt.

Beispiel: $(111{,}01)_2 = (11101)_2\times2^{-2} = 29\times2^{-2} = \frac{29}{4}$

3.2.5.1 Festkommadarstellung (Fixed Point Representation)

Zur Darstellung der Zahlen in der Festkommadarstellung werden diese in einem Wort der Länge n als Dualzahl geschrieben. Das Komma steht dabei verabredungsgemäß an einer festen Stelle, z. B. n = 16. Im folgenden werden zur Vereinfachung wieder positive Zahlen angenommen.



f_{15} f_0
MSB LSB
Most Significant Bit Least Significant Bit

Steht das Komma links vom MSB, dann sind nur Brüche (< 1) darstellbar.
Steht das Komma rechts vom LSB, dann sind nur ganze Zahlen (Abstand = 1) darstellbar.

Beispiele:

Bei n = 16 und Komma rechts vom LSB können dann ganze Zahlen in dem Bereich:
0 bis 2^{16}-1 (0 bis 65535)
in Schritten von 1 dargestellt werden.

Bei n = 16 und Komma links vom MSB können dann gebrochene Zahlen in dem Bereich:

$$0 \text{ bis } 1-2^{-16} \quad \left(0 \text{ bis } 1-\frac{1}{65536}\right)$$

dargestellt werden.

Übung:

Welcher Bereich kann bei n=16 und Komma zwischen f_7 und f_8 dargestellt werden?

3.2.5.2 Gleitkommadarstellung (Floating Point Representation)

Die Zahl 0,000 000 000 000 000 000 001 073 wäre im Festkommaformaten mit gebräuchlicher Stellenanzahl nicht darstellbar. Dazu benötigt man die Gleitkommadarstellung: $1{,}073 \cdot 10^{-24}$.

Sei z eine positive rationale Zahl, dann kann z in die Form:

$$z = m \cdot B^e$$

gebracht werden (z: Gleitkommazahl, m: Mantisse, B: Basis, e: Exponent).

Für B = 2 gilt die Darstellung:

$$z = m \cdot 2^e$$

Zusätzlich kann zur eindeutigen Darstellung m eingeschränkt werden:

$$\frac{1}{2} \le m < 1$$ ("Normalisierungsbedingung“ für B = 2)

Der Mantissenwert m hat dann in der Binärdarstellung die Form m = 0,1xxxxx (Ziffer vor dem Komma = 0, 1. Ziffer nach dem Komma immer = 1). Entspricht eine Mantisse nicht dieser Darstellung, kann dies kann durch Kommaverschiebung und Ändern der Potenz erreicht werden.

Übung: Eine Zahl z (Basis B=2) sei in nicht normalisierter Form dargestellt:

$$z = (0{,}00010011)_2 \times 2^2$$

Stellen Sie die Zahl in normalisierter Form dar, so dass die Normalisierungsregel ½ ≤ m < 1 gilt.

Die obige Normalisierungsbedingung der Mantisse für eine allgemeine Basis B lautet:

$$\frac{1}{B} \le m < 1.$$

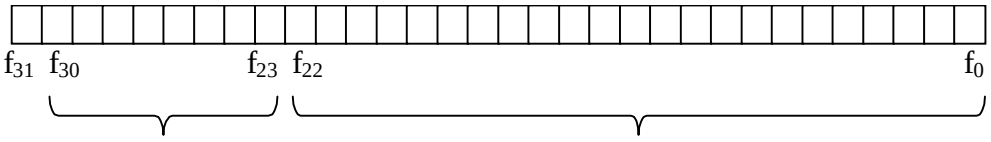
Die Mantisse m hat dann die Form m = 0,a_{-1}xxxxx (1. Ziffer a_{-1} nach dem Komma ≠ 0). Ist diese Bedingung für eine Zahl z nicht erfüllt, kann dies auch im allgemeinen Fall durch Kommaverschiebung und Ändern der Potenz e erreicht werden.

Die Normalisierungsbedingung kann auch anders lauten. Das IEEE-Format P754 (Basis B=2) normiert die Mantisse wie folgt:

$$1 \le m < 2.$$

Mit dieser Bedingung wird die Stelle links neben dem Komma immer zu 1: z=1,xxxx e^{xxx}.

Darstellung von Gleitkommazahlen im Rechner: (einfache Genauigkeit, n = 32)



VZ: Vorzeichen der Mantisse, d. h. Vorzeichen der Zahl

Es wird vereinbart, dass das höchste Bit der Mantisse (Bit links des Kommas beim IEEE-Format P754) verabredungsgemäß nicht dargestellt wird. Man verliert dadurch keine Information, da dieses Bit aufgrund der Normalisierungsbedingung immer 1 ist.

Zur Darstellung des Exponenten stehen 8 Bit zur Verfügung. Damit nicht das Vorzeichen des Exponenten dargestellt werden muss, wird statt des Exponenten e die Zahl $E = e+2^{k-1}-1 = e+127$ (k = Anzahl der Bits zur Darstellung des Exponenten) dargestellt (Excess-127 Darstellung).

Beispiel:

$5{,}75 = (101{,}11)_2 = (1{,}0111)_2 \times 2^2 \Rightarrow m=(1{,}0111)_2, e=2$

Exponent: $E=e+127=2+127=129=(10000001)_2$

5.75: 0 10000001 01110000000000000000000

Größte Zahl (theoretisch)

0 11111111 11111111111111111111111

$z = (1{,}111...1)_2 \times 2^{128} = (1+1-2^{-23}) \times 2^{128} \approx 2^{129} \approx 6{,}8 \times 10^{38}$

Meist wird jedoch eine Zahl mit einem Exponent E, der nur 1en enthält, als Überlauf gewertet.

Größte Zahl (praktisch)

0 11111110 11111111111111111111111 $\approx 3{,}4 \times 10^{38}$

Kleinste Zahl (praktisch)

1 11111110 11111111111111111111111 $\approx -3{,}4\ 10^{38}$

Kleinste positive Zahl (praktisch)

0 00000001 00000000000000000000000 $\approx 1{,}17\ 10^{-38}$

$z = 1{,}0 \times 2^{1-127} = 2^{-126} \approx 1{,}17 \times 10^{-38}$

Exponent E mit nur 0 ist meist zur Darstellung der 0 reserviert.

Genauigkeit der Mantisse:

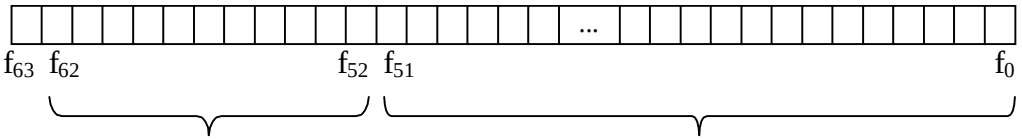
Der Abstand zweier benachbarter Mantissenwerte beträgt 2^{-23}. Soll eine zwischenliegende Zahl mit dieser Mantissengenauigkeit dargestellt werden, so muss ihr Wert auf einen benachbarten Mantissenwert gerundet werden. Dabei ist ein Fehler von maximal dem halben Abstand der darstellbaren Mantissenwerte unvermeidlich. Der Fehler beträgt somit:

$1/2 \times 2^{-23} = 2^{-24} \approx 10^{-7}$.

Übertragen auf das Dezimalsystem entspricht dies einer Genauigkeit von 7 Dezimalstellen hinter dem Komma.

Doppelte Genauigkeit:

Zur Erweiterung der Genauigkeit kann eine größere Wortlänge verwendet werden. Diese Darstellung bezeichnet man mit „doppelte Genauigkeit“. Eine Zahl belegt dann 8 Byte = 64 Bit im Speicher:



Die zusätzlichen Stellen werden zwischen dem Exponenten und der Mantisse aufgeteilt.

Der Exponent besteht nun aus 11 Bit, der Wert E errechnet nun sich zu E = e+1023 (Excess-1023 Darstellung).

Die Mantisse besteht nun aus 53 Bit. Damit erhält man eine Genauigkeit in der Darstellung der Mantisse von: $1/2 \times 2^{-53} = 2^{-54} \approx 5 \times 10^{-17}$.

Als weitere Eckdaten ergeben sich:

Größte Zahl $z = (1{,}111...1)_2 \times 2^{2046-1023} \approx 2 \times 2^{1023} = 2^{1024} \approx 1{,}7 \times 10^{308}$

Kleinste Zahl $\approx -1{,}7 \times 10^{308}$

Kleinste positive Zahl $z = 1{,}0 \times 2^{1-1023} = 2^{-1022} \approx 2{,}2 \times 10^{-308}$

3.2.5.2.1 Addition von Gleitkommazahlen

Gegeben seien zwei Zahlen z_1 und z_2:

$$z_1 = m_1 \cdot 2^{E_1} \quad , \quad z_2 = m_2 \cdot 2^{E_2}$$

Es gelte für die Exponenten: $E_2 < E_1$. Dann läßt sich z_2 in nicht-nomalisierter Form darstellen:

$$z_2 = m_2 \cdot 2^{E_2} = m_2 \cdot 2^{E_2 - E_1 + E_1} = \frac{m_2}{2^{E_1 - E_2}} \cdot 2^{E_1}$$

Dann errechnet sich die Summe aus den beiden Zahlen z_1 und z_2 wie folgt:

$$z_1 + z_2 = m_1 \cdot 2^{E_1} + \frac{m_2}{2^{E_1 - E_2}} \cdot 2^{E_1} = \left(m_1 + \frac{m_2}{2^{E_1 - E_2}} \right) \cdot 2^{E_1}$$

Vor Abspeichern des Ergebnisses muss die errechnete Mantisse $(m_1 + m_2/(2^{E_1 - E_2}))$ durch Schieben des Kommas wieder normalisiert werden, so dass die Normalisierungsbedingung wieder gilt (z.B. $1 \le m < 2$ beim IEEE-Format P754). Zur Kompensation des Schiebens ist der Exponent entsprechend anzupassen.

Beispiel: $z_1 = 0{,}101 \times 2^{10}$, $z_2 = 0{,}110 \times 2^3$, Normalisierungsregel: $½ \le m < 1$

$z_1 + z_2 = 0{,}101 \times 2^{10} + 0{,}110 \times 2^{3-10+10}$

$= 0{,}101 \times 2^{10} + \frac{0{,}110}{2^7} \cdot 2^{10}$

$= (0{,}101 + 0{,}0000000011) \times 2^{10}$

$= \underline{\underline{0{,}101000011 \times 2^{10}}}$ Ergebnis ist bereits normalisiert.

Beispiel: $z_1 = 0{,}1111 \times 2^{10}$, $z_2 = 0{,}1000 \times 2^9$, Normalisierungsregel: $½ \le m < 1$

$z_1 + z_2 = 0{,}1111 \times 2^{10} + 0{,}1000 \times 2^{9-10+10}$

$= 0{,}1111 \times 2^{10} + \frac{0{,}100}{2^1} \cdot 2^{10}$

$= (0{,}1111 + 0{,}0100) \times 2^{10}$

$= 1{,}0011 \times 2^{10}$ Ergebnis noch nicht normalisiert.

$= \underline{\underline{0{,}10011 \times 2^{11}}}$

3.2.5.2.2 Multiplikation von Gleitkommazahlen

Gegeben seien zwei Zahlen z_1 und z_2:

$$z_1 = m_1 \cdot 2^{E_1} \quad , \quad z_2 = m_2 \cdot 2^{E_2}$$

Dann errechnet sich das Produkt der beiden Zahlen wie folgt:

$$z_1 \cdot z_2 = m_1 \cdot m_2 \cdot 2^{E_1} \cdot 2^{E_2} = (m_1 \cdot m_2) \cdot 2^{(E_1 + E_2)}$$

Vor Abspeichern des Ergebnisses muss auch hier die errechnete Mantisse $(m_1 \cdot m_2)$ durch Schieben des Kommas normalisiert werden, so dass die Normalisierungsbedingung wieder gilt (z.B. $½ \le m < 1$). Zur Kompensation des Schiebens ist der errechnete Exponent $(E_1 + E_2)$ entsprechend anzupassen.

Beispiel: $z_1 = 0{,}101 \times 2^{10}$, $z_2 = 0{,}110 \times 2^3$, Normalisierungsregel: $½ \le m < 1$

$z_1 \cdot z_2 = 0{,}101 \times 2^{10} \cdot 0{,}110 \times 2^3$

$= 0{,}01111 \times 2^{13}$

$= \underline{\underline{0{,}1111 \times 2^{12}}}$

Grundlagen der Programmierung

Vorlesungsskript der Fachhochschule Osnabrück

Prof. Dr. T. Gervens, Prof. Dr.-Ing. B. Lang, Prof. Dr.-Ing. C. Westerkamp,
Prof. Dr.-Ing. J. Wübbelmann

4 Einführung in C

4.1 Wissenswertes über C

- C ist eng mit UNIX verbunden, 90% von UNIX ist in C geschrieben
- Entstanden aus den ersten UNIX-Sprachen (ca. 1970) aus der Sprache B
  Entwickler: Ken Thompson
  Dennis Ritchie
- C ist eine vollständig typisierte Programmiersprache. Dies bedeutet, dass jedes Datenobjekt einem eindeutigen Typ zugeordnet werden muss (z.B. Zeichen, Ganze Zahl, Gleitkommazahl, ...). Dieser Typ bestimmt, welche Werte diese Variable annehmen darf.
- Standard von C war lange Zeit das von Kernighan/Ritchie herausgegebene Buch "The C Programming Language".
- 1988 wurde ANSI C veröffentlicht

4.2 Erste Beispiele

```
int main()
{
}
```

Dieses Programm besteht aus einer Funktion namens **main**, was an den runden Klammern deutlich wird. In den geschweiften Klammen steht der Funktionsrumpf {....}, der in diesem Fall leer ist.

Die main Funktion gibt an den Aufrufer einen ganzzahligen Wert zurück, der z.B. zur Fehlerbehandlung genutzt werden kann. Es ist Konvention, bei einem fehlerfreien Ablauf den Wert 0 zurückzugeben.

Jedes C-Programm muss eine main-Funktion enthalten. Von dieser main-Funktion aus werden alle weiteren Programmteile aufgerufen, welche auch in Funktionsform dargestellt werden.

Im Grunde ist jedes C-Programm eine Aneinanderreihung und Verschachtelung von Funktionen, also ein Funktionsbaum.

Obiges Programm kann mit einem Texteditor als ASCII-Datei **`test.c`** erstellt werden. Von der Betriebssystemumgebung aus kann man den C-Compiler aufrufen, um das Quellprogramm in Maschinencode zu übersetzen.

```
gcc test.c ↵
```

Der Compiler schreibt den Maschinencode in eine Datei **`a.out`**.

Das Programm wird ausgeführt durch

```
a.out ↵
```

Mit

```
gcc test.c -o test.exe
```

schreibt der Compiler den Maschinencode in eine Datei namens **`test.exe`**.

Enthält das Programm Fehler, schreibt der Compiler diese Fehler auf den Bildschirm (Standard-Fehlerausgabe).

Obiges Programm tut überhaupt nichts. Damit im Programm etwas passiert sind im Funktionsrumpf **Anweisungen** einzugeben.

Die einfachste Anweisung ist die leere Anweisung, die auch nichts bewirkt:

```
int main()
{
   ; /* leere Anweisung */
}
```

Zwei Dinge sind jetzt zu beachten:

- Hinter jeder Anweisung steht ein **Semikolon**!!!
- **Kommentare** werden zwischen /* und */ eingeschlossen. Die Kommentarzeichen und alle Zeichen zwischen den Kommentarzeichen werden vom Compiler ignoriert. Kommentare sind nicht auf eine Zeile beschränkt.

```
/*
Dies
ist auch
ein
Kommentar
*/
```

Ein Beispiel das tatsächlich etwas bewirkt:

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

Dieses Programm bringt den Text:

```
hello, world
```

auf den Bildschirm. Dies geschieht folgendermaßen: Die Hauptfunktion ***main*** ruft die Bibliotheksfunktion ***printf*** mit dem Argument "hello, world" auf. Durch ***#include <stdio.h>*** wird die Datei ***stdio.h*** vor dem Compilieren hinzugefügt. In dieser Datei stehen notwendige Informationen zu ***printf***.
Das Schlüsselwort ***return*** mit dem Argument 0 weißt die ***main*** Funktion an, den Wert 0 an den Aufrufer zurückzuliefern.

Erweiterung des Programms:

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

Die Zeichenkette ***\n*** ist die C-Schreibweise für einen Zeilentrenner. Die Ausgabe wird dann am linken Rand der neuen Zeile fortgesetzt.

Beachte: ***\n*** repräsentiert nur ein Zeichen

Analog: ***\t*** (Tabulatorsprung)

Rufen wir die Funktion printf mit folgendem Agument auf:

```
printf ("\nh\ne\nl\nl\no\n");
```

führt dies zu folgender Ausgabe:

```
h
e
l
l
o
```

_ (←Cursor)

4.3 C Schlüsselwörter

C besitzt nur einen sehr begrenzten Umfang an definierten Schlüsselwörtern. Insgesamt gibt es nur 32 Wörter, die in der Sprache C eine festgelegte Bedeutung haben. Dieses sind:

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

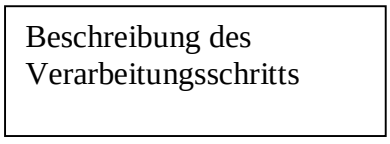
Diese Namen dürfen nur in der definierten Bedeutung benutzt werden, da der Compiler ansonsten die Übersetzung mit einem Fehler abbricht.
C unterscheidet streng zwischen Groß- und Kleinschreibung. Das bedeutet, dass z.B. ***`return`*** vom Compiler als Schlüsselwort erkannt wir, ***`Return`*** jedoch nicht.
Funktionsnamen wie ***`main`*** oder ***`printf`*** sind keine Schlüsselwörter.

4.4 Darstellung von Programmcode durch Struktogramme

Zum Entwurf eines Programms ist es hilfreich, nicht sofort mit dem Programmieren zu beginnen, sondern den Ablauf des Programms zunächst zu visualisieren. Für diesen Schritt sind **Struktogramme** sehr gut geeignet, die nach ihren Urhebern oftmals auch als **Nassi-Schneidermann-Diagramme** bezeichnet werden.

Struktogramme führen dazu, dass man ein Problem strukturiert in Teilprobleme zerlegt. Sprunganweisungen (GOTO) sind dabei nicht erlaubt. Als Mittel erlaubt die strukturierte Programmierung vielmehr die **Sequenz**, die **Iteration** und die **Selektion**. Die Elemente der Struktogramme werden im folgenden zusammen mit den zugehörigen C-Konstrukten eingeführt.

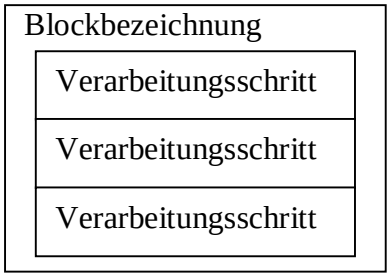
Das Grundelement des Struktogramms ist ein **Verarbeitungsschritt**, welcher als ein Rechtecksymbolisch dargestellt wird. Im Inneren des Rechtecks steht eine Beschreibung des Verarbeitungsschritts.



Sinnbild für einen
Verarbeitungsschritt

Es ist es dem Programmierer überlassen, wie komplex er seine Verarbeitungsschritte wählt. In einer Übersicht kann beispielsweise in einem einzelnen Verarbeitungsschritt der Großteil eines Programms zusammengefasst werden, in einer Feinspezifikation wird hingegen jede einzelne C-Anweisung ein Verarbeitungsschritt dokumentiert.

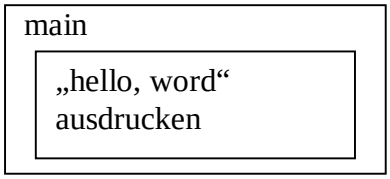
Zur Kombination mehrerer zusammenhängender Verarbeitungsschritte dient der **Block**.



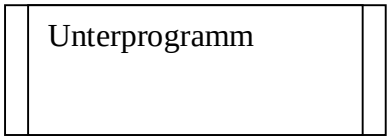
Sinnbild für einen
Block

Er kann einem Haupt- oder Unterprogramm entsprechen, aber auch einfach mehrere Verarbeitungsschritte unter einer gemeinsamen Bezeichnung zusammenfassen.

Damit kann das bisher behandelte Programm wie folgt dargestellt werden:

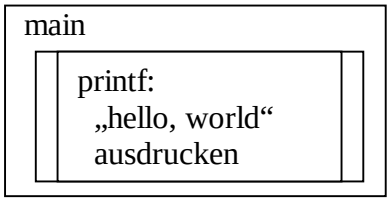


In dieser Darstellung wird nicht berücksichtigt, dass der Ausdruck durch das Unterprogramm „printf“ durchgeführt wird. Möchte man dies aufzeigen, benötigt man ein Sinnbild für einen Unterprogrammaufruf:



Sinnbild für einen
Unterprogrammaufruf

Damit lässt sich die Beschreibung des Programms noch detaillieren:



Es ist dem Programmierer beim Entwurf des Struktogramms überlassen, welchen Detaillierungsgrad er für die Beschreibung wählt. Der Detaillierungsgrad sollte so gewählt werden, dass die im Programm realisierten Algorithmen und ihre Implementierung verständlich werden und damit der zugeordnete C-Code dokumentiert wird.

4.5 Variablen

Variablen im Programm werden über die Vergabe eines Namens vereinbart. Für die Gültigkeit eines **Variablennamen** gilt:

- Besteht aus eine Folge von Ziffern, Buchstaben und dem Unterstrich „_“
- Maximale Länge: 31 Zeichen
- Das erste Zeichen muss ein Buchstabe oder der Unterstrich sein, keine Ziffer
- Es wird zwischen Groß- und Kleinschreibung unterschieden (case sensitive)
- Reservierte Wörter wie **`if`**, **`else`**, **`for`**, **`while`** dürfen nicht verwendet werden.
- Der Name muss eindeutig sein.

Zulässige Namen sind z. B.: **`a`**, **`B`**, **`SUM`**, **`x2`**, **`x_a`**, ...

Nicht zulässig: **`1a`**, **`a-wert`**

Jeder Variablen ist ein **Datentyp** zugeordnet (**`int, float, double, char`**), welcher festlegt, wie der Wert der Variablen im Speicher als Bitmuster abgelegt wird.

int: Ganzzahliger Wert (mit VZ)
Darstellung im 2er-Komplement
Länge: üblicherweise 32 Bit (auch 16 Bit oder 64 Bit)
Die Länge ergibt sich aus der Datenwortbreite des Prozessors.

float: Endliche Dezimalzahlen im Gleitkommaformat (mit VZ)
Länge: 32 Bit

double: endliche Dezimalzahl im Gleitkommaformat (mit VZ)
Länge 64 Bit (ANSI-C macht keine Vorgaben zur Länge. **double** muss nur größer oder gleich **float** sein).

char: Darstellung eines Zeichens, 8 Bit
Mit dem char-Wert kann auch gerechnet werden. Der ASCII-Code des Zeichens wird dann als Dualzahl interpretiert.

Alle Variablen in C müssen, bevor sie benutzt werden, deklariert/definiert werden. Dabei wird der Typ festgelegt. Variablendeklarationen haben folgende Form:

```
Datentyp v1, v2, v3;
```

Beispiele:

```
int i, j;
float Summe, Nenner;
double Celsius, Grad;
char Zeichen, S;
```

Neben den Grundtypen ***char***, ***int***, **float** und **double** gibt es die Qualifier

short und **long**

bzw.

signed und **unsigned**,

so dass sich folgende Typen definieren lassen:

Typ	Länge in Bytes	Wertebereich	Bedeutung	Bemerkungen
short int oder **short** **unsigned short int**	2 2	-2^{15}... $2^{15}-1$ 0 ... $2^{16}-1$	Ganzzahl mit Vorzeichen Ganzzahl ohne Vorzeichen	1, 3
int **unsigned int**	4	-2^{31}... $2^{31}-1$ 0 ... $2^{32}-1$	wie **short** (ANSI)	2, 3
long int oder **long** **unsigned long int**	4 4	-2^{31}... $2^{31}-1$ 0 ... $2^{31}-1$	Ganzzahl mit Vorzeichen Ganzzahl ohne Vorzeichen	1,2,3
float	4	$\pm 3.4*10^{-38}$ ...$\pm 3.4*10^{38}$	Gleitkommazahl (24+8 Bits), 7 Stellen	3
double **long double**	8 16	$\pm 1.7*10^{-308}$ ... $\pm 1.7*10^{308}$ $\pm 3.4*10^{-4932}$ ...$\pm 1.1*10^{4932}$	Gleitkommazahl (53+11 Bits), 15 Stellen Gleitkommazahl (65+15Bits) 30 Stellen	2,3
char **unsigned char**	1 1	-128...127 0 ... 255	ASCII-Zeichen bzw. 8 Bit-Ganzzahl mit Vorzeichen ohne Vorzeichen	3

Bemerkungen:
1. **long int** und **short int** können mit **long** bzw. **short** abgekürzt werden.

2. Die tatsächliche Anzahl von Bits variiert je nach Rechnersystem.
   Vorsicht bei der Portierung von Programmen!
3. Mit der Funktion **`sizeof`** kann die tatsächliche Speicherplatzgröße eines Datentyps in Byte ermittelt werden, z.B.:

```
int a,b;
a=sizeof(unsigned int);
b=sizeof(a);
```

Beispiele:

```
int i=5, gamma=3;
unsigned short int use;
long k;
float a, s, d, f;
char a, zeichen;
```

4.6 Konstanten

Ganzzahlige Konstanten

- Gewohnte Schreibweise **`0`**, **`-23`**, **`25786`**.
- Sie werden im Speicher im **`int`-Format** abgelegt.
- **`long`**-Konstanten werden am Ende mit dem Buchstaben l oder L geschrieben.
- **`Unsigned`** (vorzeichenlose) Konstanten werden am Ende mit u oder U geschrieben.
- Ganzzahlige Konstanten können auch **oktal (`0...`)** oder **hexadezimal (`0x...`)** oder **(`0X...`)** geschrieben werden.

Konstanten für Gleitpunktzahlen

- Dezimal oder Exponentialform: **`12.34`**, **`-1E3`**, **`.25`**, **`7E+25`**.
  Achtung: Im deutschen Sprachgebrauch wird eins Komma zwei verwendet, in C trennt jedoch der Punkt die Vor- und Nachkommastellen.
- Die Darstellung im Speicher erfolgt im **`double`-Format**.
- Typ **`float`** wird mit der Endung f oder F bezeichnet.

Char-Konstanten

- In einfachen Hochkommata eingeschlossene Zeichen wie **`'x'`**, **`'O'`**, **`'5'`**,**`'\n'`**, **`'\0'`**.
  Man beachte **`'1'`** ist nicht **`1`**!
- Die Darstellung im Rechner erfolgt als ASCII-Zeichen.

Sonderzeichen für char-Konstanten

`'\n'`	Zeilenvorschub
`'\f'`	Seitenvorschub
`'\t'`	Tabulatorsprung
`'\b'`	Rücktaste
`'\\'`	Schrägstrich
`'\''`	einfaches Hochkomma
`'\"'`	doppeltes Hochkomma
`'\ooo'`	Zeichencode in oktaler Darstellung
`'\xhh'`	Zeichencode in hexadezimaler Darstellung

Beispiele:

Konstante	Art	intern
195	dezimale Ganzzahl	**int**
195l	dezimale Ganzzahl	**long int**
195u	dezimale Ganzzahl	**unsigned int**
0303	oktale Ganzzahl	**int**
0XA3	hexadezimale Ganzzahl	**int**
0XFUL	hexadezimale Ganzzahl	**unsigned long int**
1.25	Gleitkomma	**double**
1.25f	Gleitkomma	**float**
'D'	1 Zeichen	**char**
'\x41'	1 Zeichen	**char**
'\101'	1 Zeichen	**char**

4.7 Ein- und Ausgabe von Zeichen

Die Standardbibliothek **stdio.h** enthält zwei weitere wichtige Funktionen zur Ein- und Ausgabe über Tastatur:

getchar und **putchar.**

getchar liefert bei jedem Aufruf das nächste Zeichen vom Eingabe-Zeichenstrom als Funktionswert. Nach der Anweisung:

```
int c;
c=getchar( );
```

enthält die Variable c das nächste Zeichen. Die Eingabezeichen werden in der Regel von der Tastatur gefordert. Auch Sonderzeichen wie Zeilenumbruch, Tabulator, Ende der Eingabe werden wie einzelne Zeichen behandelt.
Alle gültigen Zeichen, die von **getchar** geliefert werden, lassen sich in einem Byte kodieren und könnten somit in einer Variablen vom Typ **char** gespeichert werden. Das Zeichen "Ende der Eingabe" (End Of Transmission/**EOT**), welches mit **Control-d** auf der Tastatur ausgelöst wird, bewirkt, dass **getchar** danach nur noch den Wert **EOF** liefert. **EOF** ist eine symbolische Konstante, welche das Ende des Eingabestroms signalisiert. Sie ist nicht als Zeichencode vereinbart. Sie wird daher zusätzlich zu den gültigen Zeichencodes als Wert im **int**-Bereich vereinbart. Üblicherweise hat die Konstante **EOF** den Wert -1.
Mit der Anweisung:

```
int c;
putchar (c);
```

wird das Zeichen, dessen Code in der der Variablen c gespeichert ist, am Bildschirm ausgegeben.

Beispiel: Kopierprogramm

```
#include <stdio.h>
int main()
{
   int c;
   c=getchar ();
   while (c!=EOF) {
      putchar (c);
      c=getchar ();
   }
   return 0;
}
```

4.8 Arithmetische Operationen

a+b	Addition
a-b	Subtraktion
a*b	Multiplikation
a/b	Division
a%b	Rest bei ganzzahliger Division (Datentypen: **int**) z. B.: **20%3** = 2
±a	Vorzeichen

Rangordnung der Operationen

+, -	Vorzeichen
***, /, %**	Punktrechnung
+, -	Strichrechnung

Vor Ausführung einer arithmetischen Operation finden Typanpassungen (englisch: Casting) statt, wenn die Operanden von unterschiedlichem Typ sind. Es wird immer vom "niedrigen" in den "höheren" Typ umgewandelt:

char → **int** → **float** → **double**

Vorsicht: Das Ergebnis hat immer den gleichen Typ wie die (angepassten) Operanden. Beim nachfolgenden Beispiel

```
int n;
double x;
n=5;
x= 3/n;
```

erhält somit x den Wert 0!

4.9 Mathematische Funktionen

sqrt(x)	$\sqrt{x}$
exp(x)	e^x
log(x)	ln(x)
log10(x)	$\log_{10}(x)$
pow(x,y)	x^y
sin(x)	sin(x)
cos(x)	cos(x)
tan(x)	tan(x)
asin(x)	arcsin(x)

acos(x)	arccos(x)
atan(x)	arctan(x)
fabs(x)	|x|
sinh(x)	sinh(x)
cosh(x)	cosh(x)

Damit der Compiler diese Funktion findet, ist die entsprechende Informationsdatei **math.h** im Vorfeld zu laden durch:

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

Als Argumente können Variablen oder auch arithmetische Ausdrücke an die Funktionen übergeben werden. Das Ergebnis der gelisteten Funktionen ist in jedem Fall vom Typ **double**!! Dies muss unbedingt bei der Weiterverarbeitung oder der Ausgabe berücksichtigt werden, sonst entstehen falsche Ergebnisse.

Beispiel:

```
#include <math.h>
...
double x=1.0, z;
z=cos(x*x+tan(x));
```

Wichtig: Ein Programm mit mathematischen Funktionen muss mit

```
gcc datei.c -lm
```

übersetzt und gelinkt werden. Hierdurch wird dem Compiler mitgeteilt, in welcher Bibliothek der Objektcode der mathematischen Funktion zu finden ist.

4.10 Wertzuweisung

Die Wertzuweisung in C erfolgt mit dem Operator "=". Die Wertzuweisung hat die folgende Form:

```
Variable=Ausdruck;
```

Der Datentyp des Ausdrucks wird nach der Berechnung umgewandelt in den Datentyp der Variablen!
Erfolgt beispielsweise eine Zuweisung eines Gleitkomma-Ausdrucks auf eine Ganzzahl-Variable, dann werden bei der Umwandlung von **float** oder **double** nach **int** alle Nachkommastellen weggelassen und nur der ganzzahlige Anteil wird der Variablen zugewiesen.
Die Zuweisung endet mit einem Semikolon.

Beispiel: Berechnung der Differenz zwischen sin(x) und $x - \frac{x^3}{3!}$

```
#include <stdio.h>
#include <math.h>
int main()
{
   float x;
   x=3;
   printf ("Differenz: %f \n",
           sin(x) - x + x * x *x/6);
   return 0;
}
```

oder:

```
#include <stdio.h>
#include <math.h>
int main()
{
   float x;
   double z;
   x=3;
   z= sin(x) - x + x * x *x/6;
   printf ("Differenz: %f \n",z);
   return 0;
}
```

Innerhalb der Funktion **printf** wird in obigem Beispiel ein **float** bzw. **double**-Wert ausgegeben. Die Position wird durch die Formatangabe „**%f**“ festgelegt.

4.11 Ausgabe-Anweisung mit printf

Die Anweisung

```
printf("Text");
```

schreibt den zwischen den beiden Hochkommas "...." stehenden Ausgabetext **Text** auf den Bildschirm.

Sonderzeichen wie Zeilenumbruch **\n** oder Tabulator **\t** können in den Text eingestreut werden. Ein beliebiges Zeichen (und damit auch ein nichtdruckbares Zeichen) kann durch seinen ASCII-Code angegeben werden, z.B. in hexadezimaler Form **\xhh** (zweistellige Hexadezimalzahl). Mit

```
printf ("\x07");
```

erzeugt der Rechner beispielsweise einen "Piep", (das ASCII-Zeichen BEL).

In den Ausgabetext kann der Wert von **Variablen** und/oder **Ausdrücken** eingesetzt werden. Die Position wird durch die Stellung von **Formatangaben** festgelegt.

Beispiel:

```
float x=5;
int k=4, anz;
...
anz=printf ("x=%f,  k=%d\n", sin(x), k);
printf ("%d Zeichen gedruckt.\n",anz);
```

Die allgemeine Syntax der **printf**-Funktion lautet:

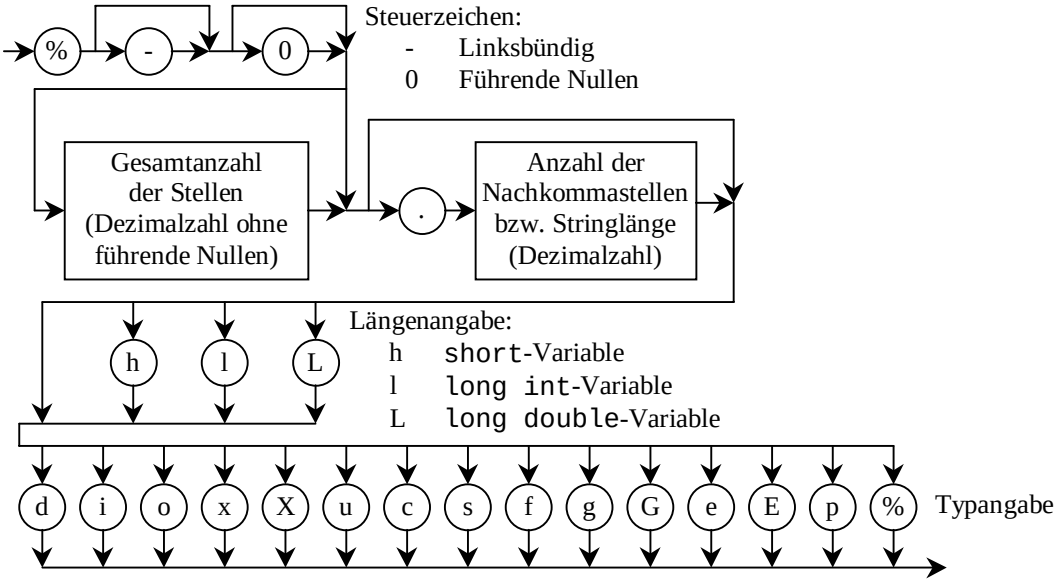
*[**int**-Variable=]* **printf***(Zeichenkette[,Variable1,Variable2,...,VariableN]);*

Optionale Elemente sind in eckigen Klammern eingeschlossen.

Argumente: Die als Argument übergebene Zeichenkette wird durch die **printf**-Funktion auf dem Bildschirm ausgegeben. In die Zeichenkette können bei der Ausgabe die Werte der nachfolgenden Variablen aus der Argumentenliste eingefügt werden. Dazu muss die Zeichenkette für jeden auszugebenden Variablenwert ein Formatfeld enthalten. Die Formatfelder werden gemäß ihrer Reihenfolge den Variablen zugeordnet. Ein Formatfeld beginnt mit einem „**%**“-Zeichen und ist weiter unten in seiner Struktur erläutert.

Rückgabewert: Der Rückgabewert der **printf**-Funktion enthält die Anzahl der ausgegebenen Zeichen.

Struktur der Formatfelder der printf-Funktion:


%
-
0
Steuerzeichen:
- Linksbündig
0 Führende Nullen
Gesamtanzahl der Stellen (Dezimalzahl ohne führende Nullen)
.
Anzahl der Nachkommastellen bzw. Stringlänge (Dezimalzahl)
h
l
L
Längenangabe:
h short-Variable
l long int-Variable
L long double-Variable
d
i
o
x
X
u
c
s
f
g
G
e
E
p
%
Typangabe

Typangabe:

d, i	Ganzzahl mit Vorzeichen, dezimal
o	Ganzzahl ohne Vorzeichen, oktale Ausgabe
x	Ganzzahl ohne Vorzeichen, hexadezimale Ausgabe. a-f in Kleinbuchstaben
X	Ganzzahl ohne Vorzeichen, hexadezimale Ausgabe. A-F in Großbuchstaben
u	Ganzzahl ohne Vorzeichen, dezimal
c	Ein einzelnes Zeichen
s	Zeichenkette (String)
f	Gleitkommazahl, in der Regel mit 6 Nachkommastellen gerundet
g, G	Gleitkommazahl in kürzester Darstellungsform, exponentiell oder fest
e, E	Gleitkommazahl in Exponentenform
p	Zeiger
%%	Ausgabe eines %-Zeichens

Beispiele:

Programmcode

```
int i=10;
float f=123.;
long int l=20;
long double d=246.8;
printf("%10d\n",i);
printf("%10.2f\n",f);
printf("%e\n",f);
printf("0x%08lx\n",l);
printf("%LE\n",d);
```

Ausgabe

```
□□□□□□□□10
□□□□123.00
1.230000e+002
0x00000014
2.468000E+002
```

Das Zeichen □ markiert ein Leerzeichen.

4.12 Eingabeanweisung mit `scanf`

Mit Hilfe der Funktion **`scanf`** können Werte für Variablen formatiert von der Tastatur eingelesen werden. Sie ist die zu **`printf`** analoge Eingabefunktion. Die allgemeine Syntax der Funktion lautet:

`[int-Variable=] scanf (`*Zeichenkette***`[,`***VarAdresse1***`,...,`***VarAdresseN***`]);`**

Optionale Elemente sind in eckigen Klammern eingeschlossen.

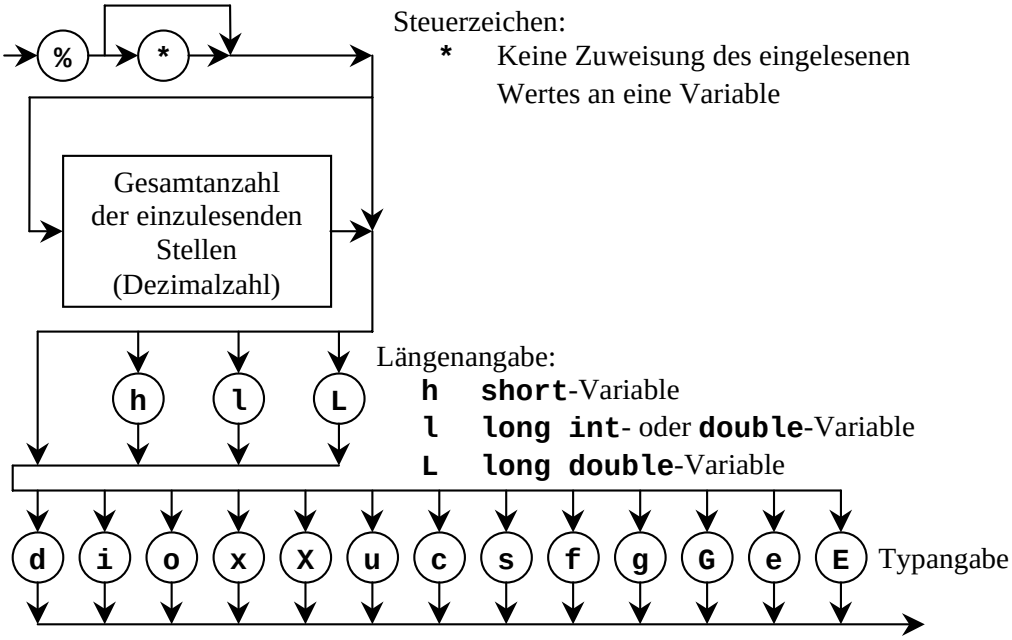
Argumente: Die als Argument übergebene Zeichenkette spezifiziert die Eingabeformate für die in der nachfolgenden Liste angeführten Variablen. Die Variablen dürfen nicht als Wert sondern müssen als **Adressen** an die **`scanf`**-Funktion übergeben werden. Damit ist es der **`scanf`**-Funktion möglich, die Inhalte der Variablen zu manipulieren.
Die Adresse einer Variablen erhält man, indem man dem Variablennamen den **Adressoperator &** voranstellt:

Variablenadresse: &Variablenname

Jeder Variablen entspricht ein Formatfeld in der Zeichenkette. Zeichen, die zu keinem Formatfeld gehören (ohne %), müssen exakt im Eingabestrom enthalten sein und werden überlesen.

Rückgabewert: Als Rückgabewert wird die Anzahl der korrekt eingelesenen Variablenwerte geliefert. Endet der Eingabestrom ohne Einlesen einer Variable, wird EOF zurückgegeben.

Struktur der Formatfelder der `scanf`-Funktion:



Die Typangaben entsprechen der Tabelle der printf-Funktion.

Beispiel: Maximum von 2 Zahlen

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

int main()
{
   double a,b,max;
   printf("Geben Sie a und b ein");
   scanf("%lf %lf",&a,&b);
   max=((a+b) + fabs(a-b))/2.0;
   printf("Maximum von %f und %f lautet %f\n",
          a,b,max);
   return 0;
}
```

Warnungen:

- Ein Vergessen des &-Zeichens führt zu Fehlern und wird beim Übersetzen nicht erkannt!!!
- Die Werte der Eingabe können durch Leerzeichen, Zeilentrenner oder Tabs getrennt sein, aber nicht durch Kommata oder andere Zeichen.
- Anzahl und Reihenfolge der Formatangaben und der Adressvariablen müssen übereinstimmen!

4.13 Vergleichsoperatoren

a<b	kleiner
a>b	größer
a<=b	kleiner oder gleich
a>=b	größer oder gleich
a==b	gleich
a!=b	ungleich

Bei einem Vergleich findet vorher wieder eine Typumwandlung wie bei der arithmetischen Operation statt.
Jeder Vergleichsausdruck besitzt den Wahrheitswert 0, falls er falsch ist, und 1 bei Wahrheit:

```
a=2;
z=(a>1);  /* z hat den Wert 1, denn a>1 */
```

Die Vergleichsoperatoren binden schwächer als die arithmetischen Operatoren. Der Ausdruck:

```
5.0 * x+7 < y/4
```

entspricht beispielsweise dem folgenden Ausdruck mit überflüssigen Klammern:

```
((5.0 * x)+7) < (y/4)
```

Vergleichsausdrücke werden vor allem als Bedingungsausdrücke in **if**- oder **for**-Kontrollstrukturen (siehe weiter unten) eingesetzt:

```
#include <stdio.h>
#include <math.h>
int main()
{
   double x;
   scanf("%lf", &x);
   if (x>0)
      printf("ln(x)=%lf\n",log(x));
   return 0;
}
```

4.14 Kontrollstrukturen

4.14.1 Sequenz, Verbundanweisung

Die Sequenz (auch als Verbundanweisung bezeichnet) ist eine Zusammenfassung von Anweisungen, welche nacheinander durchgeführt werden. Eine Sequenz kann überall dort eingesetzt werden, wo eine einfache Anweisung gefordert wird.

```
{
   <Anweisung 1>;
   <Anweisung 2>;
   ...
   <Anweisung n>;
}
```

Anweisungen im Inneren der Sequenz können beispielsweise eine Zuweisung, ein Funktionsausdruck oder eine Kontrollstruktur sein.
Im Struktogramm kann für eine Verbundanweisung ein Block eingefügt werden oder einfach eine Sequenz von Verarbeitungsschritten:

Verarbeitungsschritt 1
Verarbeitungsschritt 2
Verarbeitungsschritt 3

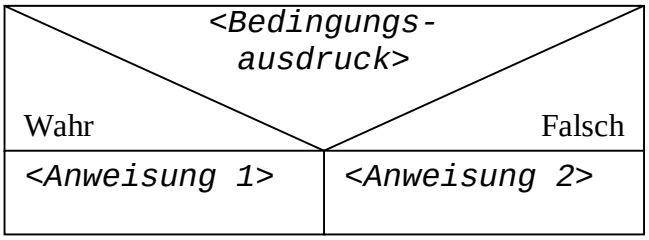
4.14.2 Einfache if-else Abfrage

Darstellung:

```
if ( <Bedingungsausdruck> )
```

```
      <Anweisung 1>;
   else
      <Anweisung 2>;
```

Anweisung 1 wird nur dann ausgeführt, wenn die Bedingung erfüllt ist. Die Bedingung ist erfüllt, wenn der Bedingungsausdruck einen Wert ≠ 0 liefert.
Liefert der Bedingungsausdruck den Wert 0, wird der **`else`**-Zweig und damit Anweisung 2 ausgeführt. Der **`else`**-Zweig kann fehlen.
Im Struktogramm wird die **`if-else`** Abfrage wie folgt dargestellt:

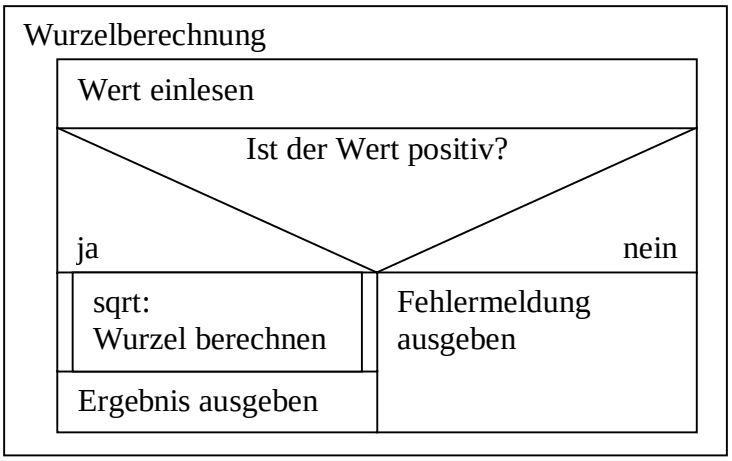


Durch Verwendung einer Sequenz statt einer einzelnen Anweisung können auch mehrere Anweisungen bei Eintreten der Bedingung und im **`else`**-Zweig ausgeführt werden:

```
if ( <Bedingungsausdruck> ){
   <Anweisung a>;
   <Anweisung b>;
   ...
} else {
   <Anweisung k>;
   <Anweisung l>;
   ...
}
```

Beispiel: Wurzelberechnung

Struktogramm:



Programm-Listing:

```
#include <stdio.h>
#include <math.h>
int main ()
{
   double x;
   scanf ("%lf", &x);
   if (x>=0)
   {
      x=sqrt(x);
      printf ("%f", x);
   }
   else
   {
      printf ("x ist negativ \n");
   }
   return 0;
}
```

Geschachtelte **if**-Anweisungen:
Das folgende Beispiel zeigt eine Mehrdeutigkeit auf, die bei geschachtelten **if**-Anweisungen auftritt und durch eine Zusatzregel aufgelöst wird:

```
...
if (a==0)
   if (b>c)
      printf("%d\n",b);
   else
      printf("%d %d\n",a,c);
...
```

Da bei der **if**-Anweisung der **else**-Zweig fehlen darf, ist es im Beispiel nicht eindeutig, ob der gezeigte **else**-Zweig zur ersten oder zur zweiten **if**-Anweisung gehört. Gehört er zur ersten **if**-Anweisung, dann hat die zweite **if**-Anweisung keinen **else**-Zweig. Im zweiten Fall hat die erste **if**-Anweisung keinen **else**-Zweig.
Die Auflösung dieser Mehrdeutigkeit schafft eine Zusatzregel in C, die einen **else**-Zweig dem benachbarten **if** zuordnet.
Will man diese Regel umgehen, muss man explizit Klammern setzen und damit die gewünschte Zuordnung herstellen. Die Zuweisung des **else**-Zweigs zum ersten **if** erhält man beispielsweise wie folgt:

```
...
if (a==0) {
   if (b>c)
      printf("%d\n",b);
} else
   printf("%d %d\n",a,c);
...
```

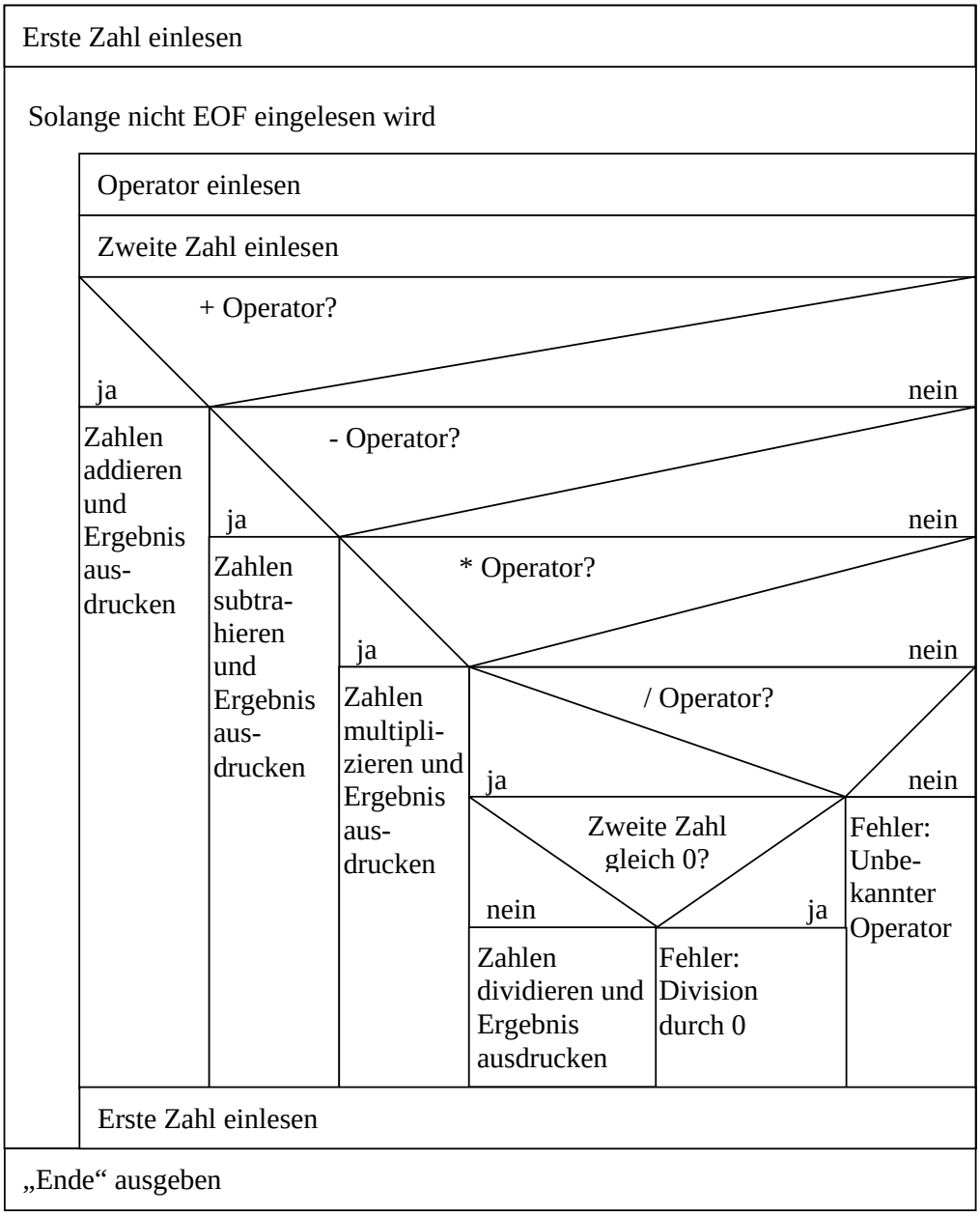
4.14.3 Mehrfachabfrage - else if

Darstellung:

```
if ( <Bedingung 1> )
   <Anweisung 1>;
else if ( <Bedingung 2> )
   <Anweisung 2>;      /* ... */
else if ( <Bedingung n> )
   <Anweisung n>;
else <Anweisung n+1>;
```

Nur die erste Anweisung, deren Bedingung erfüllt ist, wird ausgeführt. Ist keine Bedingung erfüllt, so wird, falls vorhanden, die letzte **`else`**-Anweisung (n+1) ausgeführt.
Als Anweisung kann auch eine Verbundanweisung mit einer Sequenz von Anweisungen in geschweiften Klammern ausgeführt werden.

Beispiel: Taschenrechner



Programm-Listing:

```
int main()
{
   float x,y;
   char operator;

   printf("\nGib eine Zahl ein >");


   while (EOF!=scanf("%f",&x))
   {
      printf("\nGib einen Operator ein (+/-) >");
      getchar();
      operator=getchar();
      printf("\nGib noch eine Zahl ein >");
      scanf("%f",&y);
      if (operator=='+')
         printf("\ndas Ergebnis: %f",x+y);
      else if (operator=='-')
         printf("\n das Ergebnis: %f",x-y);
      else printf("ungueltige Eingabe :%c",operator);
      printf("\nGib eine Zahl ein >");
   }
   printf("\nEnde\n\n");
   return 0;
}
```

4.14.4 Mehrfachabfrage - switch

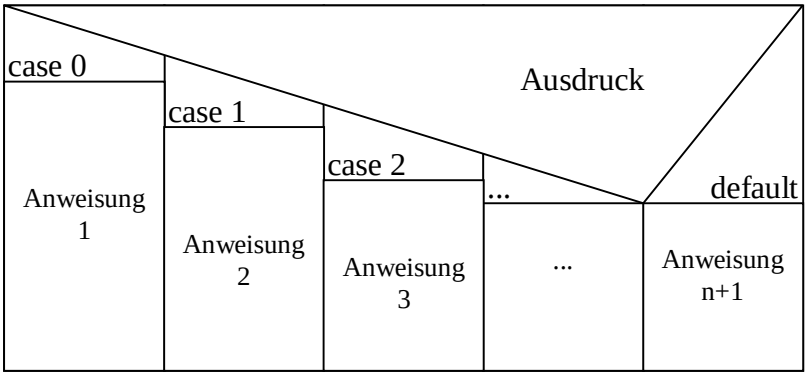
Darstellung:

```
switch ( <Ausdruck> )
{
   case <konstanter Ausdruck 1>:
      <Anweisung 1>;
      break;
   case <konstanter Ausdruck 2>:
      <Anweisung 2>;
      break;
   ...
   case <konstanter Ausdruck n>:
      <Anweisung n>;
      break;
   default:
      <Anweisung n+1>;
}
```

Zunächst wird der Wert des Ausdrucks in der **switch**-Zeile errechnet. Dieser Wert wird mit den Werten *<konstanter Ausdruck i>* der **case**-Zeilen von oben beginnend verglichen. Jeder konstante **case**-Ausdruck darf nur einmal vorkommen. Stimmt der Wert von *<Ausdruck>* mit einem konstanten **case**-Ausdruck überein, werden alle dieser **case**-Zeile nachfolgenden Anweisungen bis zur **break**-Anweisung oder bis zum Ende des **switch**-Blocks ausgeführt. Die **break**-Anweisung sorgt dafür, dass der **switch**-Block

verlassen wird. Ist keine der Bedingungen erfüllt, so werden, falls vorhanden, die Anweisungen hinter der **`default`**-Zeile ausgeführt.

Die **`switch`**-Anweisung wird durch das nachfolgende Struktogramm visualisiert:



Beispiel:

```
#include <stdio.h>
int main()
{
   /* Vereinbarung der Variablen */
   char a;
   /* Programmcode */
   a=getchar(); /* Zeichen einlesen */
   switch (a)
   {
      case 'a':
      case 'A':
         printf("\n a oder A eingegeben. \n");
         break;
      case 'b':
      case 'B':
         printf("\n b oder B eingegeben. \n");
         break;
      default:
         printf("\n Falsche Eingabe \n");
   }
   return 0;
}
```

4.14.5 while - Schleife

Darstellung:

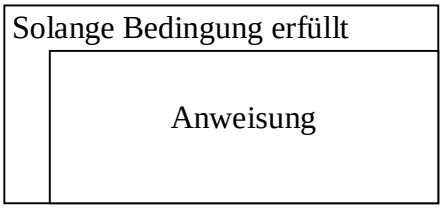
```
while ( <Bedingung> )
   <Anweisung>;
```

Die **`while`**-Schleife ist eine kopfgesteuerte Schleife. Die enthaltene Anweisung wird ausgeführt, solange die Bedingung erfüllt ist. Somit muss die Anweisung irgendwie Einfluss auf die Bedingung nehmen, sonst wird die Schleife unendlich oft ausgeführt.

Natürlich kann als Anweisung innerhalb der **while**-Schleife kann auch eine Verbundanweisung verwendet werden:

```
while ( <Bedingung> ) {
   <Anweisung 1>;
   <Anweisung 2>;
   <Anweisung n>;
}
```

Das Struktogramm für die kopfgesteuerte Schleife hat die folgende Form:



Beispiel: Bestimmung der höchsten Dezimalziffer einer natürlichen Zahl

```
#include <stdio.h>
int main()
{
   int Zahl;
   scanf("%d", &Zahl);
   while (Zahl>=10)
      Zahl=Zahl/10;
   printf("Höchste Dezimalziffer: %d\n", Zahl);
   return 0;
}
```

4.14.6 do-while - Schleife

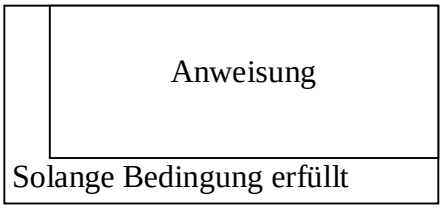
Darstellung:

```
do
   <Anweisung>;
while ( <Bedingung> );
```

Die **do-while** - Schleife ist ein fußgesteuerte Schleife. Zunächst wird die Anweisung einmal ausgeführt. Solange die Bedingung erfüllt ist, wird die Anweisung erneut ausgeführt. Als Anweisung ist auch eine Verbundanweisung möglich:

```
do
{
   <Anweisung 1>;
   <Anweisung 2>;
   <Anweisung n>;
} while ( <Bedingung> );
```

Das Struktogramm für die fußgesteuerte Schleife hat die folgende Form:



4.14.7 for - Schleife

Darstellung:

```
for ( <Ausdruck 1>; <Bedingung>; <Ausdruck 2> )
   <Anweisung>;
```

Bei der Ausführung der **for**-Schleife werden die folgenden Schritte abgearbeitet:

1. Falls vorhanden wird der *<Ausdruck 1>* ausgewertet.
2. Die *<Bedingung>* wird ausgewertet.
   Bei erfüllter (oder fehlender) Bedingung wird die *<Anweisung>* ausgeführt.
   Ist die Bedingung nicht erfüllt, wird die Schleife beendet.
3. *<Ausdruck 2>* wird ausgewertet.
4. Fortsetzung mit Punkt 2.

Beispiel: Ausdruck aller Zahlen von 0 bis 99

```
#include <stdio.h>
int main ()
{
    int i;
    for (i=0; i<100; i=i+1)
        printf ("%d\n", i);
    return 0;
}
```

Die **for**-Schleife ist gleichbedeutend mit folgendem Code-Fragment:

```
<Ausdruck 1>;
while ( <Bedingung> )
{
   <Anweisung>;
   <Ausdruck 2>
}
```

Statt einer einzelnen Anweisung kann in der for-Schleife auch eine eine Verbundanweisung angegeben werden:

```
for ( <Ausdruck 1>; <Bedingung>; <Ausdruck 2> )
{
   <Anweisung 1>;
   <Anweisung 2>;
   <Anweisung n>;
}
```

Für die **for**-Schleife gibt es kein spezielles Struktogramm. Es kann das Struktogramm der **while**-Schleife verwendet werden, wenn man als Kopfbedingung die Ausdrücke und die Bedingung der **for**-Schleife einträgt.

Bei der **for**-Schleife dürfen <*Ausdruck 1*>, <*Bedingung*> und/oder <*Ausdruck 2*> fehlen; die Semikolons müssen aber geschrieben werden. Nachfolgendes Programmfragment realisiert beispielsweise eine Endlosschleife:

```
...
for (;;)
   <Anweisung>;  /* Endlosschleife */
...
```

4.15 Steuerung von Schleifen durch `break` und `continue`

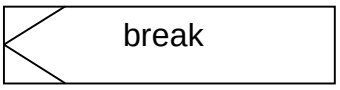
Bisher wurden die Schleifen allein durch die Kopf- oder Fußbedingungen gesteuert, der Anweisungsteil wurde als Ergebnis der Bedingung unbedingt abgearbeitet.
Zur Fortschaltung oder Beenden einer Schleife aus dem Anweisungsteil heraus dienen die **`break`**- und die **`continue`**-Anweisung.

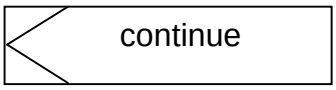
Die **`break`**-Anweisung wurde bereits im Zusammenhang mit der **`switch`**-Anweisung behandelt. Sie führte dort zu einem sofortigen Beenden der **`switch`**-Anweisung. Gleiches gilt bei Schleifen. Eine **`break`**-Anweisung führt zum sofortigen Beenden der aktuellen Schleife, ohne dass Kopf- oder Fußbedingungen geprüft werden. Die Programmausführung wird mit den Anweisungen hinter der Schleife fortgesetzt.

Beispiel: Kopierprogramm (Version 2)

```
#include <stdio.h>
int main()
{
   int c;
   while(1)
   {
      c=getchar();
      if (EOF==c)
         break;
      else
         putchar(c);
   }
   return 0;
}
```

Die **`continue`**-Anweisung bewirkt, dass der aktuelle Durchlauf des Anweisungsteils einer Schleife beendet wird. Bei **`for`**-Schleifen wird dann der *<Ausdruck 2>* ausgewertet und bei allen Schleifentypen die *<Bedingung>* der Schleife geprüft. Ist die Bedingung erfüllt, wird ein neuer Durchlauf des Anweisungsteils gestartet.
Im Struktogramm ist die Darstellung der break- und continue-Anweisung nicht fest vorgegeben. Häufig wird das folgende Symbol für beide Anweisungen verwendet, die Unterscheidung erfolgt durch Benamung:

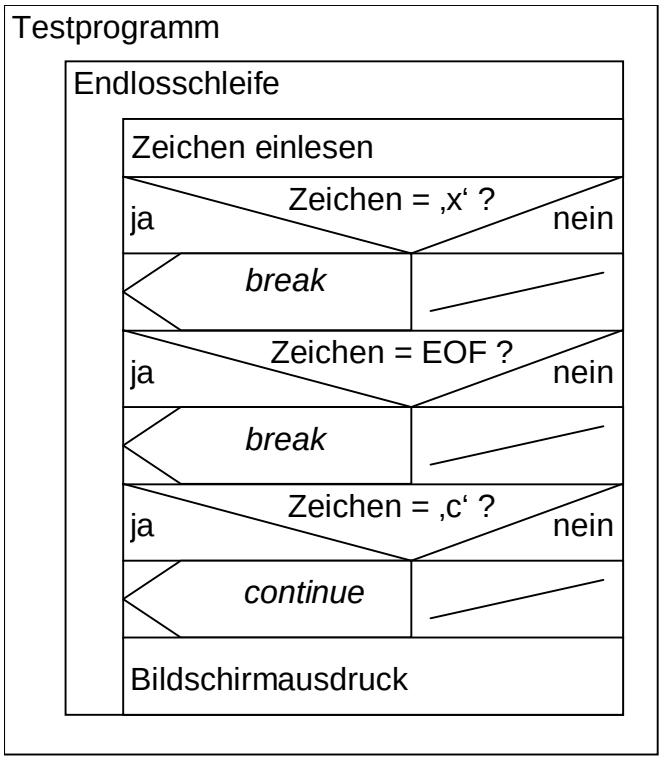




Beispiel:

```
#include <stdio.h>
int main()
{
   int c;
   for(;;)
   {
      c=getchar();
      if (c=='x') break;
      if (c==EOF) break;
      if (c=='n') continue;
      printf("Weder x noch c gedrueckt\n");
   }
   return 0;
}
```

Zugehöriges Struktogramm:



4.16 Logische Operatoren

Es gibt folgende logische Operatoren:

!	Nicht
&&	UND
||	ODER

Sie erzeugen aus Wahrheitswerten neue Wahrheitswerte. Bei Wahrheit liefern sie den Wert 1, sonst 0. Die Operanden können Zahlenwerte beliebigen Typs sein (also auch **`float`**, **`double`**). Ein Wert ≠ 0 steht für Wahrheit, ein Wert = 0 für Unwahrheit.

Die logischen Operatoren **&&** und **||** binden schwächer als die bereits bekannten Vergleichsoperatoren und arithmetischen Operatoren. Der Negationsoperator **!** bindet aber stärker. Der **&&**-Operator bindet stärker als der **||**-Operator.

Die Auswertung erfolgt von links nach rechts und wird abgebrochen, sobald das Ergebnis feststeht.

```
<Ausdruck 1> || <Ausdruck 2>
```

Ausdruck 2 wird nur dann ausgewertet, falls der Ausdruck 1 falsch ist.

```
<Ausdruck 1> && <Ausdruck 2>
```

Ausdruck 2 wird nur dann ausgewertet, falls der Ausdruck 1 wahr ist.

Beispiele:

a) `4.5 < x && x < 9.0`

liefert den Wahrheitswert 1, falls 4.5 < x < 9.0 ist.

b) `!-7.45e+4` hat den Wahrheitswert 0

c) `!a == 0` wahr, wenn a ≠ 0; gleichbedeutend mit `(!a)==0`

d) `x && y && z` ist gleichbedeutend zu `(x && y) && z`

e) `x1 || x2 && ! x3 && x4 || x5`

ist gleichbedeutend mit:

```
(x1 || ((x2 && (!x3)) && x4) || x5
```

f) `(c >= '1' && c <= '9') || (c >= 'A' && c <= 'Z')`

Wahr, falls c eine Ziffer oder ein großer Buchstabe zwischen A und Z ist.

g) `b > c && d == a || c <= a`

ist gleichbedeutend mit:

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

4.17 Inkrement und Dekrement

Die Addition und Subtraktion von 1 zu bzw. von einer Variablen ist eine sehr häufige Operation. Deshalb gibt es für diesen Zweck zwei spezielle Operatoren

`++` und `--`.

Beispiele:

`a++;` entspricht `a=a+1;`

`a--;` entspricht `a=a-1;`

Der Inkrement- und Dekrement-Operator kann sowohl vor als auch nach der Variablen stehen:

Präfix-Notation: `++a` bzw. `--a`

Postfix-Notation: `a++` bzw. `a--`

In der Präfixnotation wird a inkrementiert bzw dekrementiert bevor die Variable genutzt wird. In der Postfixnotation erfolgt die Modifikation der Variablen nach deren Nutzung.

Beispiel:

a) `a=5*(b++);` ist gleichbedeutend mit `a=5*b; b=b+1;`

b) `a=5*(++b);` ist gleichbedeutend mit `b=b+1; a=5*b;`

Ist z.B. b=1, so errechnet sich b in beiden Fällen zu 2. Die Variable a wird im ersten Fall zu 5 gesetzt, im zweiten Fall zu 10.

Inkrement und Dekrement können auf jeden Variablentyp angewendet werden. Sie binden stärker als die arithmetischen Operatoren.

Häufige Verwendung in Zählschleifen:

```
for(i=0;i<N;i++)
   ...
```

Vorsicht bei mehrfacher Anwendung dieser Operatoren:

```
a=b++  +  b++; /* Compiler-abhängig*/
```

Das Ergebnis ist Compiler-abhängig, da unklar ist, ob vor der zweiten Verwendung von b die erste Inkrementoperation schon durchgeführt wurde.

4.18 Zuweisungsoperatoren

Neben der einfachen Zuweisung kann eine Zuweisung eine mit einer arithmetischen Operation verknüpft werden. Die Beispiele zeigen, wie der Operand auf der linken Seite der Zuweisung in die arithmetische Berechnung einbezogen wird.

=	Einfache Zuweisung
+=	Addition und Zuweisung
-=	Subtraktion und Zuweisung
***=**	Multiplikation und Zuweisung
/=	Division und Zuweisung
%=	Modulo und Zuweisung

Beispiele:

```
x *=4;                  x=x*4;
y-=1;                   y=y-1;
w/=23;                  w=w/23;
jj*=xx++  + ++yy;       jj=jj*(xx++  +  ++yy);
```

4.19 Shiftoperatoren

<<	Linksverschiebung der Bitkette
>>	Rechtsverschiebung der Bitkette

Beispiel:

```
x << 3 /* Entspricht x*8 */
y >> 4 /* Entspricht x/16 */
1 << 4 /* Entspricht 2 hoch 4 */
```

Linksverschiebung:

Der Binärwert des linken Operanden (int-Typ: char, short, int, long) wird soviele Bitstellen nach links verschoben, wie der rechte Operand (immer positiv!) angibt. Die unteren Bitstellen werden in der Regel mit Nullen aufgefüllt.

Rechtsverschiebung:

Der Binärwert des linken Operanden (int-Typ: char, short, int, long) wird soviele Bitstellen nach rechts verschoben, wie der rechte Operand (immer positiv!) angibt. Die oberen Bitstellen werden in der Regel mit Nullen aufgefüllt.

Achtung: Wird ein Typ mit Vorzeichen nach rechts verschoben, so wird auf manchen Systemen das Vorzeichenbit, also 1, auf anderen Systemen 0 nachgeschoben.

4.20 Bit-Operatoren

&	bitweises UND
|	bitweises (inkl.) ODER
^	bitweises Exklusiv-ODER (XOR)
~	1-Komplement (Vertauschen 0-1/Inversion)

Bitoperationen werden genutzt, um auf eine Auswahl von Bits innerhalb einer Variablen zuzugreifen.

Beispiele:

```
short int x,y; /* 16 Bit Werte*/
y=x | 0xF000;
x=x & 077;
x=x & ~077; /* gleichbedeutend mit   x=x & 0177700 */
x=x & (1<<5) /* maskiert Bit 5 von x. */
y=x ^ ((1<<3)|(1<<5)|(1<<7)); /* invertiert Bits 3, 5 und 7 */
```

Der Unterschied zwischen **x&&&y** und **x&y** ist zu beachten. Für x=1, y=2 ist **x&&&y=1** (wahr) und **x&y=0**.

4.21 Fragezeichenoperator

Syntax:

(<Bedingung>) **?** *<Ausdruck1>* **:** *<Ausdruck2>*

Ist die *<Bedingung>* erfüllt, liefert der Operator den Wert von *<Ausdruck1>*, sonst von *<Ausdruck2>*.

Beispiel:

```
z=(a>b) ? a : b ;  /* Maximum von a und b */
```

dies entspricht dem folgenden Code:

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

Beispiel:

```
z=(a>0) ? a:-a ;  /*Absolutbetrag von a  */
```

dies entspricht dem folgenden Code:

```
if (a>0) z=a; else z=-a;
```

Man beachte, dass vor der Auswertung Typanpassungen wie bei arithmetischen Verknüpfungen stattfinden. Der Ausdruck:

```
a>0?1:2.0
```

liefert für a>0 den Wert 1 mit Datentyp **double**.

Wie sonstige Ausdrücke kann man bedingte Ausdrücke beliebig tief schachteln:

```
x>=0. ? x==0. ? 0:1:-1; /* Vorzeichen: sign(x) */
```

ist gleichbedeutend mit

```
x>=0. ? (x= =0.? 0:1):-1
```

4.22 Bindungsstärken von Operatoren

Nachfolgende Tabelle gibt eine vollständige Übersicht über die Operatoren von C.

Priorität	Assoziativität	Operatoren	Bezeichnung
1	links	`()`	Funktionsaufruf
		`[]`	Feldzugriff
		`-> .`	Komponentenselektion
2	rechts	`++ --`	Inkrement, Decrement
		`! ~`	Negation (Logisch, Bitweise)
		`+ -`	Vorzeichen (Unär)
		`* &`	Dereferenzierung, Adresse
		`(type)`	Typkonversion
3	links	`* / %`	Arithmetische Operatoren
4	links	`+ -`	Arithmetische Operatoren
5	links	`<< >>`	Schiebeoperatoren
6	links	`< <= > >=`	Vergleichsoperatoren
7	links	`== !=`	Vergleichsoperatoren
8	links	`&`	Bitweises UND
9	links	`^`	Bitweises Exklusiv-Oder
10	links	`|`	Bitweises Oder
11	links	`&&`	logische Und
12	links	`||`	logisches Oder
13	rechts	`? :`	Fragezeichenoperator
14	rechts	`= += -= *= /= %= ^= &= |= <<= >>=`	Zuweisungsoperatoren
15	links	`,`	Komma-Operator

In der Tabelle sind die Operatoren gemäß ihrer Priorität angeordnet. Die Priorität gibt an, in welcher Reihenfolge die Operatoren in einem Ausdruck abgearbeitet werden. Die Operatoren verknüpfen in Reihenfolge absteigender Priorität die Operanden eines Ausdrucks.
Innerhalb einer Priorität bestimmt die Assoziativität die Abarbeitungsreihenfolge der Operatoren. Bei Rechts-Assozivität werden die Operatoren gleicher Prioritätsstufe von rechts nach links abgearbeitet, Bei Links-Assozivität entsprechend von links nach rechts.

Beispiele:

`!++a` ist gleichbedeutend mit **`!(++a)`**

Erhöhung von a um 1, anschließend logische Verneinung von ++a

`-a++` gleichbedeutend mit **`-(a++)`**

(Liefert als Wert -a, Anschließend erfolgt eine Erhöhung von a um 1)

`(a<0)?(b>4)?x:y:z` gleichbedeutend **`(a<0)?((b>4)?x:y):z`**

Achtung:
In C ist nicht festgelegt, in welcher Reihenfolge die Argumente einer Funktion ausgewertet werden. Die Anweisung

```
printf("%d, %d\n",++n,2*n);
```

ist nicht eindeutig. Besser:

```
++n;
printf("%d, %d\n",n,2*n);
```

Im Zweifelsfall und zur besseren Übersichtlichkeit: Klammern setzen!!!

4.23 Felder

Gleichartige, logische zusammenhängende Objekte können zu einem Feldern (Array, Vektor) zusammengefasst werden.

Vorteil:

- Für gleichartige Objekte wird eine einzelne, gemeinsame Variable verwendet.
- Die einzelnen Objekte sind gemeinsam ansprechbar.

Syntax:

Vereinbarung/Definition:

<Typ> <Name>`[`*<N>*`];` (N: Integer-Wert)

Aufruf:

<Name>`[`*<Index>*`]` (mit 0<=Index<=N-1)

Es werden N Speicherplätze des angegebenen Typs reserviert, die mit den Bezeichnern *<Name>*`[0]`, ..., *<Name>*`[`*<N-1>*`]` angesprochen werden können.

Beispiel:

Es wird ein Vektor mit Namen **`vekt`** bestehend aus 5 **`int`** Zahlen vereinbart, d.h. es wird ein zusammenhängender Speicherplatz für 5 **`int`**-Zahlen reserviert:

```
int vekt[5];
```

Die Elemente 0 und 1 des Vektors werden auf die Werte 6 und 4 gesetzt:

```
vekt[0]=6;
vekt[1]=4;
```

Der Vektor hat nun das folgende Erscheinungsbild:

`vekt`:	**`vekt[0]`**	**`vekt[1]`**	**`vekt[2]`**	**`vekt[3]`**	**`vekt[4]`**
	6	4	?	?	?

Bemerkungen:

- Das erste Feldelement von Vektoren hat immer den Index 0 das letzte hat den Index N-1.
- In C erfolgt beim Zugriff auf Vektorelemente **keine** Prüfung auf mögliche Bereichsüberschreitungen. In Anlehnung an obiges Beispiel wird beispielsweise mit:
  ```
  vekt[5]=7;
  ```
  der Speicherplatz hinter dem letzten Element **`vekt[4]`** des Vektors überschrieben und damit zerstört. Dies führt zu einem Fehler, der erst zur Laufzeit des Programms (hoffentlich) erkannt wird (Laufzeitfehler).

Initialisierung von Vektoren:

Die Elemente eines Vektoren können schon bei der Definition des Vektors initialisiert werden. Dazu wird eine in geschweiften Klammern eingeschlossene Liste von initialen Werten mit einem Gleichheitszeichen an die Definition angehängt:

```
<Typ> <Name>[<N>]={<Wert0>,<Wert1>,...};
```

Die Liste kann weniger Initialisierungswerte enthalten als der Vektor Elemente besitzt, dann werden die hinteren Elemente des Vektors, für die keine initialen Werte angegeben sind, zu 0 initialisiert.
Bei vorhandener Initialisierungsliste braucht weiterhin die Größe des Vektors (Dimension) nicht spezifiziert werden, dann wird die Anzahl der Elemente aus der Länge der Initialisierungsliste ermittelt:

```
<Typ> <Name>[]={<Wert0>,<Wert1>,...,<WertN>};
```

Beispiele:

Vollständige Initialisierung eines Vektors:

```
int tage[6]={31,28,31,30,27,31};
```

Anzahl der Initialisierungswerte ergibt die Dimension des Vektors:

```
int tage[]={31,28,31,30,27,31};
```

Unvollständige Initialisierungsliste für einen Vektor. Die restlichen Elemente werden mit 0 initialisiert.

```
int n[5]={1,2,3};  /* n[3]=0, n[4]=0 */
```

Folgendes Programm liest einen Vektor der Dimension 5 ein und gibt den Betrag (Länge) des Vektors aus.

```
#include <stdio.h>
#include <math.h>
int main()
{
   int i;
   float n[5],sum=0,skprod;
   for(i=0;i<5;i++)
   {
      scanf("%f",&(n[i]));
      sum=sum+pow(n[i],2);
   }
   skprod=sqrt(sum);
   printf("Betrag: %f ",skprod);
   return 0;
}
```

4.24 Zeichenketten

In C gibt es keine eigenen Datentyp für Zeichenketten (String). Zum Speichern einer Zeichenkette wird ein Vektor vom Typ char vereinbart. Im letzten Element dieses Zeichenketten-Vektors wird das **Endezeichen '\0'** abgelegt.

Beispiel:

Die folgenden alternativen Definitionen vereinbaren einen Zeichenketten-Vektor, der die Zeichenkette „hallo" speichert. Da neben den Buchstaben auch das Endezeichen abgelegt werden muss, hat der Vektor die Länge 6.

```
char text[6]={'h','a','l','l','o','\0'};
```

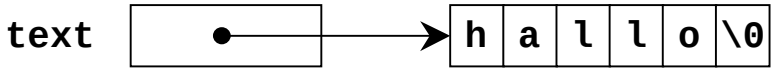
oder

```
char text[]="hallo";
```

oder

```
char *text="hallo";
```

Jede dieser alternativen Definitionen legt für den Vektor **text** einen Zeiger fest, der auf den Speicherbereich zeigt, wo die Elemente des Vektors abgelegt sind:



Nahezu alle C-Programme, die Zeichenketten verarbeiten, erwarten als Abschluss einer Zeichenkette das Endezeichen **'\0'**. Jeder Programmierer muss selbst dafür sorgen, dass seine Zeichenketten mit **'\0'** abschließen!
Werden Zeichenketten unbekannter Länge verarbeitet, so muss ein **char**-Vektor mit genügend vielen Elementen angelegt werden, um alle möglichen Zeichenketten darin abzulegen. Dieser Vektor wird mit den Zeichen der aktuellen Zeichenkette gefüllt. Das Endezeichen wird hinter das letzte gültige Zeichen der aktuellen Zeichenkette gesetzt, die hinterliegenden Zeichen können beliebige Werte enthalten.
String-Konstanten, d.h Zeichenketten in Anführungszeichen (z.B. **"WESTERKAMP"**), sind in C ebenfalls Zeichenketten-Vektoren. Bei Vereinbarung einer String-Konstante wird ein Feld ein Zeiger auf die erste Feldkomponente angelegt. Damit ist der Zugriff auf einzelne Elemente mit den bekannten Operatoren möglich:

"WESTERKAMP"[3]	entspricht	**'T'**
"LANG"[4]	entspricht	**'\0'**
***"GERVENS"**	entspricht	**'G'**
***("Lang"+2)**	entspricht	**'n'**

Beispiel Kopierfunktion, Vektorvariante

```
void strcpy(char *quelle, char *ziel) {
   int i=0;
   while((ziel[i]=quelle[i])!='\0')
      i++;
}
```

Beispiel Kopierfunktion, Zeigervariante

```
void strcpy(char *quelle, char *ziel) {
   while(*ziel++=*quelle++) ;
}
```

Verfügbare Zeichenkettenfunktionen:
In der Headerdatei **string.h** sind Deklarationen für viele Zeichenkettenfunktionen enthalten:

int strlen(char *s);	**Ermittelt Länge von s (ohne \0)**
int strcmp(char *s, char *t);	**Vergleich zweier Strings (=0: s gleich t, <0: s vor t, >0: s hinter t)**
int strncmp(char *s, char *t, int n);	**Vergleich wie strcmp aber höchstens n Zeichen**
char *strcpy(char *s,char *t);	**kopiert t nach s inkl \0, liefert s**
char *strncpy(char *s,char *t, int n);	**kopiert höchstens n Zeichen**
char *strcat(char *s,char *t);	**kopiert t an das Ende von s**
char *strchr(char *s,int t);	**Sucht das Zeichen t in s, liefert Zeiger in s**
char *strstr(char *s, char *t);	**sucht t als String in s (Substring)**

Bei Verwendung dieser Funktionen bindet der Linker den zugehörigen Code für die Funktionen in das ausführbare Programm ein.
Man beachte, dass diese Funktionen keine Speicherverwaltung übernehmen. Der Speicherbereich für vorkommende Zeichenketten muss also vorher bereitgestellt werden!

Beispiel: Einlesen einer Zeichenkette

```
char z[100]; /* maximal 100 Zeichen vorsehen */
/* Zeile einlesen bis zum ersten \n */
gets(z);
```

oder:

```
char z[100]; /* maximal 100 Zeichen vorsehen */
/* wortweise bis zum 1. white space einlesen */
scanf("%s",z);
```

Beide Funktionen fügen ein \0 an.

Beispiel: Ausgabe einer Zeichenkette

```
char z[100]="Hallo";
/* Ausgabe bis zum ersten \0, es wird ein \n angehängt: */
puts(z);
printf("%s",z);
/* Linksbündig, m Mindestbreite, n Anzahl auszudruckender Zeichen: */
printf("%-m.ns",z);
/* Es werden i Zeichen ausgedruckt: */
printf("%.*s",i,s);
```

Beispiel: Text verändern

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

int main()
{
   char z[100], umkehr[100], up[100], low[100];
   int i, laenge;

   printf("Text eingeben (max. Laenge 100)\n");
   gets(z);
   laenge=strlen(z);

   /* Grossbuchstaben */
   for(i=0; i<laenge; i++)
   {
      up[i]=toupper(z[i]);
   }
   up[laenge]='\0';
   printf("Gross:\t%s\n", up);

   /* Kleinbuchstaben */
   for(i=0; i<laenge; i++)
   {
      low[i]=tolower(z[i]);
   }
   low[laenge]='\0';
   printf("Klein:\t%s\n", low);

   /* Rueckwaerts */
   for(i=0; i<laenge; i++)
   {
     umkehr[i]=z[laenge-1-i];
   }
   umkehr[laenge]='\0';
   printf("Umkehrung:\t%s\n", umkehr);
   return 0;
}
```

4.25 Tipps und Tricks zum Lösen der Aufgaben

a) Arbeiten unter Solaris mit einem Texteditor-Fenster zum Schreiben des C-Programms und einem Komando-Fenster zum Compilieren.

b) Die letzten Befehle kann man sich mit **`h`** (history) anzeigen lassen. Mit !(Nr) kann der Befehl mit der entsprechenden Nummer wieder ausgeführt werden, ohne dass er wieder ganz eingegeben werden muss.

c) Eine weitere Eingabehilfe ist in der Verwendung der Cut, Copy und Paste Tasten gegeben. Erst einen Text markieren, dann in den Buffer kopieren und dann an gewünschte Stelle wieder einfügen.

d) Beim Compilieren kann man den Namen der zu erzeugenden Maschinencode-Datei angeben durch:

```
gcc auf1.c -o auf1.exe
```

e) Unter UNIX gibt es das Kommando
`cb` (c-beautifier)
`cb aufg1.c` liest die die Datei **`aufg1.c`** und schreibt es in gut lesbarer C-Struktur auf den Bildschim

f) Unter UNIX kann man die Standardein- und -ausgabe in eine beliebige Datei umleiten:

```
a.out > file.aus
a.out < file.ein
file.ein > a.out > file.aus
```

`a.out` liest die Eingabedaten aus **`file.ein`** und schreibt die Ausgabedaten in die Datei **`file.aus`**. Dies gilt auch für die meisten UNIX-Kommandos:
z. B.: **`cb aufg1.c > schoen1.c`**

g) Man kann auch zwei Programme gleichzeitig starten und die Ausgabe des ersten in die Eingabe des zweiten lenken (Pipe oder Pipeline):

```
auf1.exe | auf2.exe
```

Analog bei mehreren Programmen:

```
auf1.exe | auf2.exe | auf3.exe | ...
auf1.exe < File.ein| auf2.exe | auf3.exe > File.out
```

h) Im Texteditor besteht im Menüpunkt **`View`** die Möglichkeit, bestimmte Zeilennummern anzusprechen, in denen sich möglicherweise ein Fehler befindet.

i) Die Beschreibung von UNIX-Kommandos und C-Funktionen kann man sich mit **`man`** (Manual) anzeigen lassen, z. B.: **`man cb`**

j) Unter Open Windows findet man die C- und UNIX-Handbücher im Open Windows Menu Answerbook.

Grundlagen der Programmierung

Vorlesungsskript der Fachhochschule Osnabrück

Prof. Dr. T. Gervens, Prof. Dr.-Ing. B. Lang, Prof. Dr.-Ing. C. Westerkamp,
Prof. Dr.-Ing. J. Wübbelmann

5 Funktionen

Größere Programme werden aus mehreren Gründen in kleinere Einheiten aufgeteilt:

- **Übersichtlichkeit:**
  Kleinere Programmteile sind in ihrer logischen Struktur besser zu überschauen.
- **Modularität**
  Verschiedene Teile können von verschiedenen Programmierern entwickelt werden.
  Die gleiche Funktion kann von verschiedenen Stellen aufgerufen werden.
- **Verstecken von Implementierungsdetails (information hiding)**
  Es werden die Details der Implementierung versteckt, die aus übergeordenter Sicht irrelevant sind. Nur die Schnittstelle zum Aufrufen der Funktion muss bekannt sein.

Ein C-Programm besteht i.a. aus mehreren Funktionen und globalen Datendeklarationen. Diese können über mehrere Quelldateien verteilt sein, d.h. getrennte Übersetzbarkeit ist möglich.
Es werden dann alle drei Dateien übersetzt, um die ausführbare Programmdatei zu erhalten. Die Dateien können gemeinsam übersetzt und zum ausführbaren Programm zusammengebunden werden:

gcc file1.c file2.c file3.c

Der aus jeder Quelldatei erzeugte Objektcode wird in einer Objektcode wird in einer Datei abgelegt, welche im Namen weitgehend mit der Quelldatei übereinstimmt, aber statt mit „.c“ mit „.o“ endet. Gibt es in einer der Dateien, z.B. in der Datei file1.c ein Fehler, braucht nach Behebung des Fehlers nur diese neu übersetzt werden:

gcc file1.c file2.o file3.o

Von den übrigen Dateien wird direkt der Objektcode angegeben, dann geht die Erzeugung des ausführbaren Programmcodes schneller.
Die Erzeugung des Objektcodes kann für jede Datei einzeln erfolgen. Der Compilerschalter „-c“ bewirkt, dass die Quelldatei nur zur Objektdatei übersetzt wird; der Bindeprozess muss dann separat durchgeführt werden:

gcc –c file1.c
gcc –c file2.c
gcc –c file3.c
gcc file1.o file2.o file3.o –o test.exe

5.1 Funktionsdefinition, -deklaration und -aufruf

Beispiel:

```
#include <stdio.h>

int power (int m, int n);/* Funktionsdeklaration */

int main(){
   int i=5;
   /* Funktionsaufruf */
   printf ("%d\n", power(2,i)); /* Funktionsaufruf */
}

/* Funktionsdefinition */
int power (int basis, int exp) {
int i, p;
   p=1;
   for (i=1;i<=exp;i++)
      p=p*basis;
   return p;
}
```

5.1.1 Funktionsdefinition

Syntax:

```
<Funktionstyp> <Funktionsname>
      (<Parameterdeklarationen>) {
   <Vereinbarungen>;
   <Anweisungen>;
   return (<Ausdruck>);
}
```

Funktionstyp:
Jede Funktion besitzt einen Funktionstyp. Zugelassen sind alle bekannten Basistypen.
Eine Funktion, die keinen Wert zurückliefert, bekommt den Typ **`void`** (Prozeduren).
Ist kein Typ angegeben, wird als Typ **`int`** angenommen.

Funktionsnamen:
Namen werden nach den gleichen Regeln wie Variablennamen gebildet. Möglichst selbsterklärend wählen!

Parameterdeklarationen:
Der Funktion können Parameter übergeben werden. Alle Parameter sollen explizit deklariert werden. Diese Parameter werden auch Formalparameter genannt. Die runden Klammern müssen vorhanden sein.

Funktionsrumpf {...}:
Der Funktionrumpf ist ein in sich abgeschlossener Block.

Return:
Die **`return`**-Anweisung gibt den durch den Ausdruck berechneten Wert an die aufrufende Stelle zurück. Runde Klammern dürfen auch fehlen. Fehlt der Ausdruck, so wird kein Wert zurückgegeben, sondern nur zur aufrufenden Stelle zurückgesprungen. Die **`return`**-Anweisung muss nicht, sollte aber möglichst am Ende stehen.
Auch die **main-Funktion** besitzt einen Funktionstyp und eine **`return`**-Anweisung. Fehlt diese, wird implizit **`int`** als Typ angenommen und ein **`return`** eingefügt. Viele Unix-Systeme fordern, einen int-Wert zurückzugeben. Da einige Compiler bei fehlendem **`main`**-Typ eine Warnung ausgeben, ist es empfehlenswert, **`main`** als **`int`** zu deklarieren und Null (normales Programmende) zurückzugeben, wenn das Programm normal endet.

5.1.2 Funktionsdeklaration:

Vor dem ersten Aufruf muss die Funktionen deklariert werden:
Syntax:

```
<Funktionstyp> <Funktionsname>
      (<Parameterdeklarationen>);
```

Wird die Funktion vor dem ersten Aufruf definiert, kann eine zusätzliche Deklaration entfallen.

5.1.3 Funktionsaufruf:

Der Funktionsaufruf kann an jeder Stelle durch die Angabe des Funktionsnamens erfolgen, an der auch ein Wert des Datentyps stehen könnte. Beim Aufruf werden alle Werte der aktuellen Parameter in den Speicherplatz des jeweils zugehörigen formalen Parameters kopiert (**call by value)**
Syntax:

```
<Funktionsname> (<Ausdruck1>, <Ausdruck2>,
                 ..., <AusdruckN>)
```

Die Ausdrücke 1 bis N werden zunächst ausgewertet und die Werte der Ausdrücke den entsprechenden Parametern zugewiesen.
Das **Ergebnis** des Funktionsaufrufs ist (falls die Funktion nicht vom Typ **`void`** ist) die Auswertung des **`return`**-Ausdrucks.

Beispiele:

a) Hallo-Programm

```
#include <stdio.h>

void gruss (void);

int main (void) {
   gruss();
   return 0;
}

void gruss() {
   printf("Hallo");
}
```

b) Polynomberechnung

```
double poly(double, double)

int main(){
   ...
   z=poly(a,b);
   ...
}

double poly (double x, double y){
   double z;
   z=x*x+2*y*y+3*x*x;
   return z;
}
```

Durch ANSI-C wurde die Funktionsbehandlung gegenüber früheren C-Versionen verändert. Häufig sieht man Funktionsdeklarationen und -definitionen noch in der alten Version. Beispiel b) sieht in der alten Version wie folgt aus:

```
double poly(); /* andere Deklaration */

int main() {
   ...
   z=poly(a,b); /* unverändert */
   ...
}

double poly (x,y) /* andere Definition */
double x,y;
{
   double z;
   z=x*x+2*y*y+3*x*x;
   return z;
}
```

Die neue Funktionsdeklarationon macht es den Compilern leichter, Fehler bei der Parameterübergabe zu erkennen. Der alte Stil wird von den meisten Compilern noch geduldet. Wir verwenden jedoch ausschließlich den neuen Stil.

5.2 Funktionen und der Stack

Der Stapel eines Prozessors (Stack) ist ein Speicherbereich, auf dem dynamisch zu einer Funktion gehörige lokale Variablen und Informationen abgelegt werden. Die zu einer Funktion zugehörige Information wird als *Stapelrahmen* (oder besser engl. *Stackframe*) bezeichnet.

Lokale Variable sind die Variable, welche im Inneren des Funktionsrumpfs vereinbart werden (und nicht der Speicherklasse **`static`** zugeordnet sind).

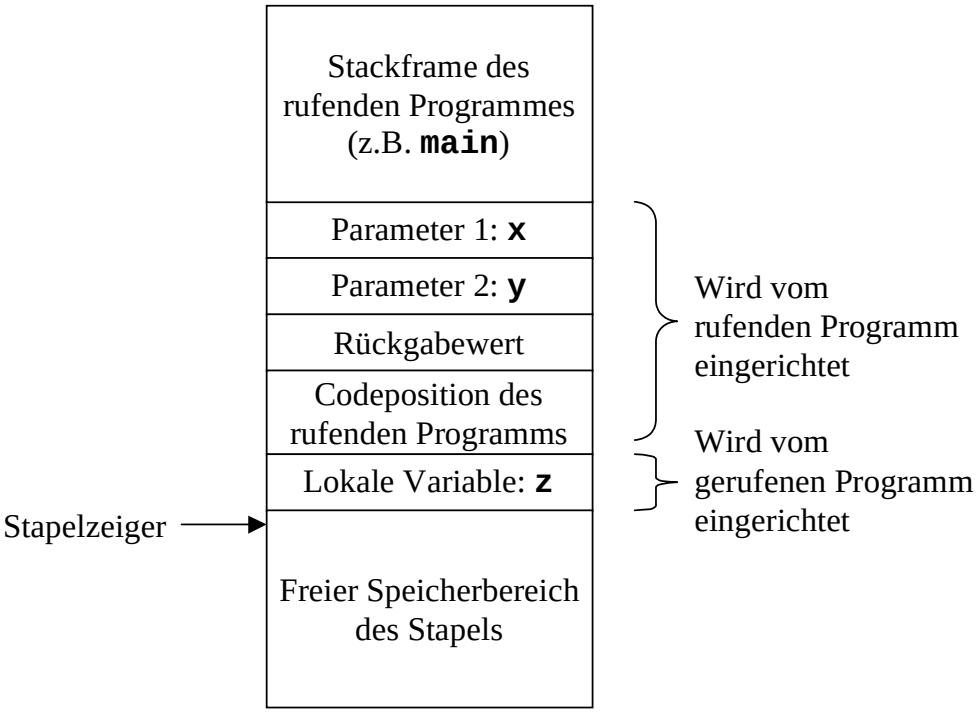
Informationen zu einer Funktion sind

- Position im Code des rufenden Programmes. Diese Position wird benötigt, damit nach Auführung der gerufenen Funktion der Prozessor wieder zur rufenden Funktion zurückkehren kann.
- Übergabeparameter (aktuelle Parameter), die von der rufenden an die gerufene Funktion übergeben werden.
- Rückgabewert der gerufenen Funktion, der an die rufende Funktion übergeben wird.

Zu einem Stapel gehört ein **Stapelzeiger**, der auf den nächsten freien Eintrag des Stapels zeigt. Relativ zum Stapelzeiger erfolgt der Zugriff auf die lokalen Variablen der Funktion und auf die Informationen zur Funktion. Der Compiler berechnet den Offset vom Stapelspeicher zur Speicherzelle, in der die gewünschte Information liegt. Die Adresse der Speicherzelle wird aus dem Inhalt des Stapelspeichers zur Laufzeit und dem zur Kompilierungszeit bekannten Offset ermittelt.

Beispiel:

Die oben verwendete Funktion „**`double poly(double,double)`“** benötigt beispielsweise den folgenden *Stackframe*:



Der Zugriff auf die Variable **`z`** (Größe 4 Bytes) erfolgt beispielsweise mit der Adresse (Wert des Stapelzeigers)+4. Benötigt man ebenfalls 4 Bytes zur Speicherung der Codeposition (Größe ist prozessorabhängig), so erfolgt beispielsweise der Zugriff auf den Parameter **`x`** mit der Adresse (Wert des Stapelzeigers)+20. Beim Beenden schreibt die Funktion **`poly`** ihren durch die **`return`**-Anweisung spezifizierten Ergebniswert in die Speicherzelle mit der Adresse (Wert des Stapelspeichers)+12.

Jedesmal wenn eine Funktion aufgerufen wird wird ein Stackframe für diese Funktion auf dem Stack angelegt. Nach Bearbeitung der Funktion wird der Stackframe durch Verändern des Wertes im Stapelzeiger wieder vom Stapel entfernt.

Werden aus einer Funktion heraus weitere Funktionen aufgerufen, so werden deren Stackframes zusätzlich auf den Stapel gepackt. Der oberste Rahmen gehört zur aktuellen Funktion, deren Anweisungen gerade vom Prozessor bearbeitet werden.

Die Stackframes liegen (beginnend mit dem Rahmen der **`main`**-Funktion) in der Reihenfolge auf dem Stapel, in der hierarchisch die Funktionen bis zur aktuellen Funktion aufgerufen wurden.

5.3 Parameterübergabe

Bei der Übergabe von Parametern an eine Funktion werden in der Sprache C *Werte* übergeben, die von der Funktion ausgewertet werden können. Diese Art der Parameterübergabe wird als **call by value** bezeichnet. Bei jedem Aufruf der

Funktion werden die Werte der aktuellen Parameter von der rufenden Funktion in die auf dem Stapel eigerichteten Speicherzellen der Parameter *kopiert*.

Zur Übergabe von Information von einer Funktion an die rufende Funktion ist nur der Rückgabewert vorgesehen. Die Werte der aktuellen Parameter können durch eine Wertveränderung der Funktionsparameter nicht beeinflusst werden.

- Vorteil dieses Vorgehens: sicher
- Nachteil: unflexibel

Andere Sprachen (z.B. Pascal) bieten die Möglichkeit, die Referenz einer Variablen an ein eine Funktion zu übergeben. Dann kann die Funktion über diese Referenz direkt auf eine Variablen des rufenden Programms zugreifen und auch deren Werte verändern. Diese Art der Parameterübergabe wird als **call by reference** bezeichnet.

In der Sprache C wird die *call by reference*-Parameterübergabe dadurch simuliert, dass man einer Funktion Adressen von Parametern übergibt. Bei *call by reference*-Parametern wird mit Hilfe des **Adress-Operators &** die Adresse des Speicherplatzes ermittelt, der einer Variablen zugeordneten ist.

Beispiel:

```
kreis(radius, &umfang, &flaeche);
```

Eine Funktion mit Namen **`kreis`** wird aufgerufen. Von den drei Parametern wird der erste Parameter **`radius`** als Wert und die beiden Parameter **`umfang`** und **`flaeche`** als Adresse übergeben.

Zum Zugriff auf den Wert einer Speicherzelle, von der die Adresse verfügbar ist, dient der **Inhalts-Operator `*`**. Er wird auch als **Dereferenzierungsoperator** bezeichnet.

Beispiel:

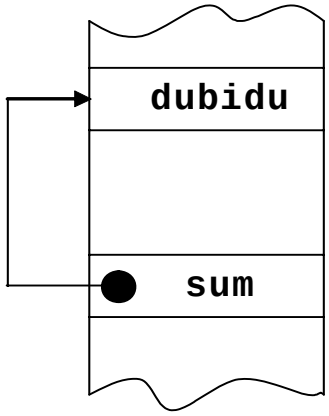
Der Ausdruck **`*&summe`** (Inhalt von der Adresse von **`summe`**) ist gleichbedeutend mit dem direkten Zugriff auf eine Variable mit Namen **`summe`**.

Adressen:

- sind **vorzeichenlose Dualzahlen** einer bestimmten vom Prozessor abhängigen Länge (z.B. 32 Bit).
- Adressen werden in C als **Zeiger** bezeichnet (da die Adresse immer auf den Anfang eines Speicherbereichs zeigt).
- Zeiger der verschiedenen Datentypen werden in C unterschieden. Grund ist, dass die Längen und Informationsdarstellungen der Datentypen unterschiedlich sind und bei Kenntnis des Datentyps immer korrekt auf den Inhalt der zugeordneten Speicherzelle zugegriffen werden kann. Zu jedem Datentyp gibt es somit einen zugehörigen **Zeigertyp** (Zeiger auf int, Zeiger auf float, usw.).
- Die **Definition von Zeigervariablen** erfolgt über den zugehörigen Datentyp, dem Variablennamen eines Zeigers wird jedoch das Zeichen **`*`** vorangestellt.

Beispiele:

```
float  *z_float;   /* Zeiger auf float.    */
int a, *b, *c, d;  /* b,c Zeiger auf int   */
float dubidu,*sum; /* sum Zeiger auf float */
...
sum=&dubidu;
```



- Ein **dereferenzierter Zeiger** wird wie eine Variable des Datentyps verwendet, auf den der Zeiger zeigt. (z.B. Wertzuweisung).

  Beispiel:

```
double *a, b;
...
a=&b;
*a=5;
(*a)++;  /* b=6 */
```

Beispiele:

a) Vertauschen der Werte von zwei Variablen

```
int tausch(int *, int *);

int main(){
   int a=5,b=6;
   tausch (&a,&b);
   return 0;
}

int tausch(int *x, int *y){
   int hilf;
   hilf=*x;
   *x=*y;
   *y=hilf;
   return 0;
}
```

b) Berechung der Fläche und des Umfangs eines Kreises

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

void kreis(double, double*, double*);

int main() {
   double radius=10,flaeche,umfang;
   kreis(radius, &flaeche, &umfang);
   printf("Radius=%f Flaeche=%f Umfang=%f",
           radius,flaeche,umfang);
}

void kreis(double r, double *f, double *u){
   *u=2*r*M_PI;
   *f=M_PI*r*r;
}
```

5.4 Rekursion

Eine C-Funktion darf direkt oder indirekt sich selber wieder aufrufen. Solch ein eigener Aufruf wird als **Rekursion** bezeichnet. (Unabhängig von der verwendeten Programmiersprache muss bei rekursiven Funktionsaufrufen durch den Algorithmus gewährleistet werden, dass die Rekursion terminiert.)

Beispiel: Berechnung der Fakultät n!

```
int fak (int n){
   if (n)
      return n*fak(n-1);
   else
      return 1;
}
```

Die Funktion **`fak`** wird (n+1)-mal mit den folgenden Parameterwerten aufgerufen:

n, n-1, n-2, n-3, ..., 1, 0.

Bei jedem Aufruf wird die Abarbeitung durch den erneuten Aufruf von fak realisiert, bis bei n=0 der Rückgabewert 1 feststeht:

`fak(n)` ⇄ `fak(n-1)` ⇄ ... ⇄ `fak(1)` ⇄ `fak(0)`

Iterative Lösung:

```
int fak(int n){
   int i,f=1;
   for (i=1;i<=n;i++)
      f*=i;
   return f;
}
```

Rekursion empfiehlt sich bei oft wiederholenden gleichen Operationen oder bei rekursiv definierten Datenstrukturen (Bäume etc.).
Jede rekursive Lösung kann umgeformt und in eine iterative Lösung überführt werden. Diese führt jedoch in den meisten Fällen zu einem komplizierteren Programmcode. Die rekursive Realisierung eines Algorithmus benötigt hingegen mehr Speicherplatz als die iterative Lösung, ist aber übersichtlicher.

Beispiel: Zahlen einlesen bis eine 0 eingegeben wird. Dann die Zahlen in umgekehrter Reihenfolge wieder ausgeben.

```
void f(void){
   int i;
   scanf("%d",&i);
   if(i)
      f();
   printf("%d\n",i);
}
int main(void){
   f();
   return 0;
}
```

Beispiel: **Berechnung des ggT** nach dem euklidischen Algorithmus
(ggT: größter gemeinsamer Teiler)

Definition: d heißt ggT(a,b) falls
1. d/a und d/b
2. c/a, c/b => c/d

Beispiele:
ggT(12,30)=6

Wichtige Eigenschaften des ggT:
ggT(0,a)=ggT(a,0)=a
ggT(a,b)=ggT(a-qb,b) mit a-qb>0.

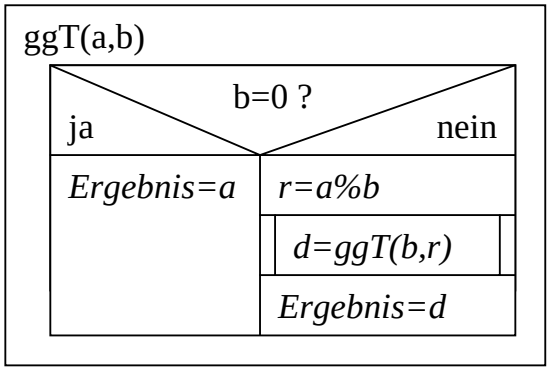
Beispiele:
ggT(114,18) = ggT(114-18,18) = ggT(96,18)
=ggT(96-18,18)=...
=ggT(96%18,18)
=ggT(6,18)= ggT(18,6)
=ggT(6,0)=6

Allgemein gilt für den ggT:
ggT(a,b)=ggT(b,r) mit r=a%b

Algorithmus in Pseudocode:

```
ggT(a,b):
  Falls b=0 {
    /* weitere Rekursion nicht notwendig */
    Ergebnis=a
  } sonst {
    Ermittle Rest r der Division a/b: r = a%b
    Berechne d=ggT(b,r) /* rekursiver Aufruf */
   Ergebnis ist d
  }
```

Algorithmus als Struktogramm:



Beispiel:

a	B	r
114	18	114-6*18=6
18	6	18-3*6=0
6	0	

Beispiel: **Türme von Hanoi**

Problem: In der Stadt Hanoi stehen drei Türme. Auf einem dieser Türme befinden sich 64 Scheiben mit wachsendem Durchmesser. Buddhistische Mönche befassen sich einer Legende nach mit der Aufgabe, diese Scheiben auf eine andere Säule zu tragen unter den Nebenbedingungen

1. Niemals darf mehr als eine Scheibe bewegt werden
2. Nie darf eine größere Scheibe auf einer kleineren liegen.

Die Mönche glauben, das Ende der Welt würde durch die Lösung angekündigt. Dieses scheinbar komplexe Problem besitzt eine einfache rekursive Lösung. Angenommen das Problem ist für N-1 Scheiben (N: Anzahl der Scheiben) bereits gelöst, dann kann die Lösung für das Problem N Scheiben von Turm A nach C zu bringen wie folgt gelöst werden:

1. Ist N=1, bringe die eine Scheibe von A nach C. (STOP)
2. Bringe die oberen N-1 Scheiben mit Hilfe von Turm C von A nach B.
3. Bringe die letzte Scheibe von A nach C.
4. Bringe die N-1 Scheiben von B mit Hilfe von Turm A nach C.

Allgemein lässt sich zeigen, dass für N Scheiben 2^N-1 Umlegungen nötig sind. Im Fall N=64 ergeben sich somit 2^{64}-1 Bewegungen. Benötigt man pro Bewegung eine Sekunde, so sind die Mönche 584,9 Milliarden Jahre beschäftigt.

```
#include <stdio.h>
void transport(int anzahl, int von, int hilfe,
               int nach, int *p_umleg);
int main(){
    int n;       /* Anzahl der Scheiben          */
    int umleg=0; /* Anzahl der Umlegaktionen     */
    void transport(int,int,int,int, int *);
    printf("Tuerme von Hanoi\n");
    printf("Anzahl der Scheiben eingeben:\n");
    scanf("%d",&n);
    transport(n, 1, 2, 3, &umleg);
    printf("%d Umlegaktionen\n",umleg);
    return 0;
}
void transport(int anzahl, int von, int hilfe,
int nach, int *p_umleg){
    if(anzahl>1)
       transport(anzahl-1,von,nach,hilfe,p_umleg);
    printf("Bringe Scheibe %2d von %2d nach %2d\n",
            anzahl, von, nach);
    (*p_umleg)++;
    if(anzahl>1)
       transport(anzahl-1,hilfe,von,nach,p_umleg);
}
```

Für n=3 ergibt sich folgende Ausgabe:

```
Anzahl der Scheiben eingeben:
3
Bringe Scheibe  1 von  1 nach Turm  3
Bringe Scheibe  2 von  1 nach Turm  2
Bringe Scheibe  1 von  3 nach Turm  2
Bringe Scheibe  3 von  1 nach Turm  3
Bringe Scheibe  1 von  2 nach Turm  1
Bringe Scheibe  2 von  2 nach Turm  3
Bringe Scheibe  1 von  1 nach Turm  3
7 Umlegaktionen
```

6 C-Präprozessor

Vor dem eigentlichen Übersetzen werden vom C-Präprozessor einige Vorfunktionen ausgeführt. Diese werden im C-Programm mit dem Zeichen **#** markiert.

6.1 Einfügen von Dateien

Enthält der Quellcode eines C-Programms eine **`#include`**-Anweisung in der folgenden Form mit spitzen Klammern (**`'<'`**, **`'>'`**):

```
#include <Dateiname>
```

dann wird vom Präprozessor die Datei mit Namen *`Dateiname`* im Standard-Include-Verzeichnis des Compilers (z.B. `/usr/include`) gesucht und in den Quellcode eingefügt. Weitere Suchverzeichnisse können durch die Compileroption *-I* eingegeben werden; z.B.:

gcc -I /usr/wuebbelm/include test.c

Alternativ kann der Dateiname in doppelte Hochkomma eingeschlossen werden:

```
#include "Dateiname"
```

Dann wird die Datei mit Namen *`Dateiname`* zunächst im aktuellen Arbeitsverzeichnis gesucht. Wird sie dort nicht gefunden, werden die Include-Verzeichnisse wie oben beschrieben nach der Datei durchsucht.
Durch die **`#include`**-Anweisung referenzierte Dateien werden als Include-Dateien oder auch als Header-Dateien bezeichnet.

Include-Dateien können in beliebiger Schachtelungstiefe über **`#include`**-Anweisungen weitere Include-Dateien referenzieren.

Vereinbarungsgemäß besitzen Include-Dateien die Endung „.h“ (Header-Dateien). Bekannte Headerdateien sind z.B.: *stdio.h*, *math.h*, *limits.h*.

In Include-Dateien werden allgemeine Deklarationen (keine Anweisungen, kein ausführbarer Code) eines Programmpaketes zusammengefasst und mit der **`#include`**-Anweisung in alle betroffenen C-Dateien eingefügt. So ist sichergestellt, dass alle C-Dateien auf dieselben Deklarationen zugreifen.

6.2 Textersatz, Makros

C kennt einen einfachen Mechanismus, um Konstanten, d.h. festen Zeichenketten, Namen zu geben.
Syntax:

```
#define NAME zeichenkette
```

In der gesamten Quelldatei wird dann *`NAME`* durch *`zeichenkette`* ersetzt.
Beispiel:

```
#define LENGTH 13
int main(){
    printf("Laenge: %d\n",LENGTH);
```

```
    return 0;
}
```

Der Textersatz wird nur für vollständige, abgeschlossene Namen durchgeführt. Findet sich *NAME* innerhalb einer längeren Zeichenkette, findet keine Ersetzung statt.

Die Definition *NAME* wird als **Makro** bezeichnet.

Konvention: Namen der mit **#define** vereinbarten Konstanten sollten vereinbarungsgemäß nur mit GROSSBUCHSTABEN bezeichnet werden.

Textersatz ist besonders bei Konstanten nützlich:

```
#define EOF          -1
#define NULL          0
#define BUFFERSIZE 1000
```

Ein Makro kann auch parametriert werden. Das entsprechende Präprozessor-Makro hat dann folgende Form:

```
#define NAME(Arg1,...,ArgN) zeichenkette
```

Beim Aufruf des Makros *NAME* ersetzen die aktuellen Makro-Parameter jedes Vorkommen der formalen Makro-Parameter in der Zeichenkette *zeichenkette*.

Beispiel:

```
#define ISLOWER(c)  ((c)>='a' && (c) <='z')
#define TOUPPER(c)  ((c)+'A' - 'a')
#define TOLOWER(c)  ((c)+'a' - 'A')
#define max(A,B)    ((A)>(B)?(A):(B))
```

TOUPPER('x') wird zu: **(('x')+'A' - 'a')**
max(x,y) wird zu: **((x)>(y)?(x):(y))**

Ein mit **#define** definiertes Makro *NAME* kann mit:

```
#undef NAME
```

wieder gelöscht werden.

6.3 Bedingte Übersetzung

Der Präprozessor kann mit Bedingungen gesteuert werden, die bestimmen, welche Anweisungen in den Quellcode eingefügt werden.

Struktur:

```
#if Bedingung
   Anweisungen (C-Anweisungen oder Präprozessor-Direktiven)
#elif Bedingung
   Anweisungen
...
#else
   Anweisungen
#endif
```

Es werden nur die C-Anweisungen übersetzt bzw. die Präprozessor-Direktiven aus dem Abschnitt ausgeführt, bei dem die *Bedingung* erfüllt ist.

Beispiel:

```
#if  SYSTEM == SYSV
   #define HD  "sysv.h"
#elif SYSTEM == MSDOS
   #define HD "msdos.h"
#else
   #define HD "default.h"
#endif
```

Präprozessor-Bedingungen lassen sich auch mit **#ifdef** und **#ifndef** formulieren. Diese Präprozessor-Anweisungen dienen zur Überprüfung, ob ein Name definiert ist.

Beispiel:

```
#ifndef  HD
   #define HD "default.h"
#endif
```

Beispiel:

```
#define DEBUG 1
...
#ifdef DEBUG
   printf("%d\n",n);
#endif
```

Mit der Compiler-Option *-DDEBUG=1* kann das Makro auch beim Compilieren definiert werden.

gcc -DDEBUG=1 test.c

Beispiel:

```
#include <stdio.h>
main() {
#if DEBUG==1
   printf("Jetzt debuggen mit Option 1\n");
#elif DEBUG==2
   printf("Jetzt debuggen mit Option 2\n");
#endif
}
```

Grundlagen der Programmierung

Vorlesungsskript der Fachhochschule Osnabrück

Prof. Dr. T. Gervens, Prof. Dr.-Ing. B. Lang, Prof. Dr.-Ing. C. Westerkamp,
Prof. Dr.-Ing. J. Wübbelmann

7 Dateizugriff

Zur Eingabe von großen Datenmengen an ein Programm und zur langfristigen Sicherung von Daten eines Programms ist die Ein- und Ausgabe von Daten mittels Dateien unerlässlich.

Bevor man auf eine Datei zugreifen kann, muss sie zunächst geöffnet werden. Anschließend kann man solange auf die Datei zugreifen, bis sie wieder geschlossen wird. Nachfolgend sind wichtige Funktionen aus der *stdio*-Bibliothek zum sequentieller Bearbeiten von Dateien aufgelistet:

`fopen`	Datei öffnen
`fclose`	Datei schließen
`feof`	Überprüfung auf Dateiende (End of File)
`fflush`	Zwischenpuffer in Datei wegschreiben
`fgetc`	Einzelnes Zeichen aus Datei lesen
`fread`	Feld von Zeichen aus Datei lesen
`fgets`	Zeichenkette aus Datei lesen (terminiert mit '\0')
`fscanf`	Formatierte Eingabe aus einer Datei
`fputc`	Einzelnes Zeichen in Datei schreiben
`fwrite`	Feld von Zeichen in Datei schreiben
`fputs`	Zeichenkette in Datei schreiben (terminiert mit '\0')
`fprintf`	Formatierte Ausgabe in eine Datei
`fseek`	Dateizeiger verschieben
`rewind`	Dateizeiger zurücksetzen
`ftell`	Dateizeiger abfragen

Beim Aufruf von **`fopen`** prüft das Betriebssystem, ob man berechtigt ist auf die angegebene Datei zuzugreifen. Wird das Öffnen der Datei vom Betriebssystem erlaubt, erzeugt **`fopen`** eine Kontrollstruktur **`FILE`**, in der alle Informationen über die geöffnete Datei verwaltet werden. Dazu gehört der **Dateizeiger**, der die Position des nächsten zu lesenden oder zu schreibenden Datenbytes in der Datei angibt. Zeigt der Dateizeiger hinter das letzte Byte der Datei, so bewirkt ein Lesen der Datei eine **`EOF`**-Meldung. Beim Schreiben werden in diesem Fall die Datenbytes hinten an die Datei angehängt und damit die Datei erweitert.

Der Dateizeiger wird durch jeden Lese- oder Schreibvorgang um die Anzahl der gelesenen oder geschriebenen Bytes nach vorne bewegt. Somit erfolgt ein sequentieller Dateizugriff. Soll von diesem sequentiellen Zugriff abgewichen werden, kann der Dateizeiger mit **`rewind`** und **`fseek`** direkt bewegt werden. Zusätzlich kann mit **`ftell`** die Position des Zeigers abgefragt werden.

Diese Struktur vom Typ **`FILE`** ist in der Include-Datei „stdio.h“ deklariert. Beim Öffnen einer Datei richtet die Funktion **`fopen`** eine Variable vom Typ **`FILE`** ein, füllt sie mit den zu Datei gehörigen Werten und liefert einen Zeiger auf diese **`FILE`**-Struktur als *return*-Parameter zurück.

7.1 Wichtige Dateifunktionen

7.1.1 Funktion `fopen`

Beschreibung:

Öffnet eine Datei zum Lesen oder zum Schreiben.

Syntax:

```
#include <stdio.h>
FILE *fopen(char *name, char *modus);
```

Bedeutung der Parameter:

`name:` Zeiger auf Zeichenkette, welche den Dateinamen enthält.

`modus:` Zeiger auf Zeichenkette, welche die Zugriffsart auf die Datei beschreibt (Mode-String). Diese kann die Zeichen **`r`**, **`w`**, **`a`**, **`+`**, **`b`** und **`t`** mit folgender Bedeutung enthalten:

- **`r`** Öffnen ausschließlich für Leseoperationen
- **`w`** Erzeugung für Schreiboperationen. Wenn eine Datei dieses Namens bereits existiert, wird sie überschrieben.
- **`a`** Erzeugung für Schreiboperationen oder, falls die Datei bereits existiert, Öffnen für anfügende Schreiboperationen am Dateiende.
- **`r+`** Öffnen einer bereits existierenden Datei für Lese- und Schreiboperationen
- **`w+`** Erzeugung einer neuen Datei für Lese- und Schreiboperationen. Wenn eine Datei dieses Namens bereits existiert, wird sie überschrieben.
- **`a+`** Öffnen einer Datei für Leseoperationen und anfügende Schreiboperationen am Dateiende. Falls die Datei noch nicht existiert, wird sie erzeugt.

Soll eine Datei im Text-Modus geöffnet oder erzeugt werden, wird **`t`** an den String angehängt oder eingefügt.

Beispiele: **`"rt"`**, **`"w+t"`**, **`"rt+"`**

Soll eine Datei im Binär-Modus geöffnet oder erzeugt werden, wird **`b`** an den String angehängt oder eingefügt.

Beispiele **`"wb"`**, **`"a+b"`**, **`"rt+"`**

Diese Modi sind Betriebssystemsabhängig. POSIX definiert ausschließlich den Binärmodus **`b`**, allerdings ohne Funktion (Kompatibilität zu ISO C). **`t`** wird z.B. unter Windows benutzt.

Rückgabewert:

Im Erfolgsfall liefert **`fopen`** einen Zeiger auf eine gültige **`FILE`**-Struktur zurück. Bei Fehlern wird ein **`NULL`**-Zeiger als *return*-Wert zurückgegeben.

7.1.2 Funktion `fclose`

Beschreibung:

Schließt eine vorher geöffnete Datei.

Syntax:

```
#include <stdio.h>
int fclose(FILE *file);
```

Bedeutung der Parameter:

`file:` Zeiger auf gültige **`FILE`**-Struktur

Rückgabewert:

Bei Erfolg wird der Wert 0 zurückgegeben, ansonsten der Wert EOF.

Bemerkung:

Beim Terminieren eines Programms werden alle Dateien, die im Programm geöffnet aber nicht geschlossen wurden, automatisch vom Betriebssystem geschlossen.

7.1.3 Funktion `feof`

Beschreibung:

Diese Funktion überprüft, ob der Dateizeiger am Ende der Datei steht.

Syntax:

```
#include <stdio.h>
int feof(FILE *file);
```

Bedeutung der Parameter:

`file:` Zeiger auf gültige **`FILE`**-Struktur

Rückgabewert:

Wenn Dateizeiger innerhalb der Datei steht: 0.
Wenn Dateizeiger das Dateiende erreicht hat: Ungleich 0.

7.1.4 Funktion `fflush`

Beschreibung:

Bewirkt das physikalische Schreiben von gepufferten Daten in die zugeordnete Datei.
`fflush` sollte nur auf Ausgabedatenströme angewendet werden.

Syntax:

```
#include <stdio.h>
int fflush(FILE *file);
```

Bedeutung der Parameter:

`file:` Zeiger auf gültige **`FILE`**-Struktur

Rückgabewert:

Bei fehlerfreier Ausführung: 0
im Fehlerfall: EOF

7.1.5 Funktion `fgetc`

Beschreibung:

Liest ein Zeichen aus einer Datei.

Syntax:

```
#include <stdio.h>
int fgetc(FILE *file);
```

Bedeutung der Parameter:

`file:` Zeiger auf gültige **`FILE`**-Struktur

Rückgabewert:

Bei fehlerfreier Ausführung: Das gelesene Zeichen, nachdem es in einen vorzeichenlosen Integer-Wert umgewandelt wurde.

Bei Dateiende oder einem Fehler: EOF

7.1.6 Funktion `fread`

Beschreibung:

Lesen mehrerer Datenobjekte aus einer Datei.

Syntax:

```
#include <stdio.h>
size_t fread(void *buffer, size_t size,
             size_t number, FILE *file);
```

Bedeutung der Parameter:

`buffer:`	Speicherbereich zum Einlesen von Daten aus der Datei. Die Größe diese Bereichs muss mindestens **`size`*****`number`** betragen.
`size:`	Größe eines einzelnen zu lesenden Datenobjekts
`number:`	Anzahl der zu lesenden Datenobjekte
`file:`	Zeiger auf gültige **`FILE`**-Struktur

Rückgabewert:

Anzahl der gelesenen Objekte der Größe **`size`**.

7.1.7 Funktion `fgets`

Beschreibung:

Liest eine mit '\0' terminierte Zeichenkette aus einer Datei.

Syntax:

```
#include <stdio.h>
char *fgets(char *buffer, int maxlength,
            FILE *file);
```

Bedeutung der Parameter:

buffer	Zeiger auf Puffer zur Aufnahme der Zeichenkette.
maxlength	Größe des Puffers.
file	Zeiger auf gültige **FILE**-Struktur.

Rückgabewert:

Bei erfolgreichem Lesen: Zeiger auf Puffer.
Bei Fehler oder **EOF**: **NULL**-Zeiger.

buffer muss mindestens **maxlength** groß sein!

7.1.8 Funktion `fscanf`

Beschreibung:

Formatiertes Lesen aus einer Datei.

Syntax:

```
#include <stdio.h>
int fscanf(FILE *file, const char *format, ...);
```

Bedeutung der Parameter:

file	Zeiger auf gültige **FILE**-Struktur.
format	Format-String (entsprechend **scanf**).
...	Parameteradressen entsprechend dem Format-String.

Rückgabewert:

Anzahl der eingelesenen Parameter.

Vorsicht beim Einlesen von Zeichenketten: Der Aufrufer muss für genügend Speicher für den String garantieren! Meist ist **fgets()** die bessere Alternative, um Strings einzulesen.

7.1.9 Funktion `fputc`

Beschreibung:

Schreiben eines Zeichens in eine Datei.

Syntax:

```
#include <stdio.h>
int fputc(int character, FILE *file);
```

Bedeutung der Parameter:

file	Zeiger auf gültige **FILE**-Struktur.
character	Zu schreibendes Zeichen.

Rückgabewert:

Bei fehlerfreier Ausführung: Das geschriebene Zeichen.
Im Fehlerfall: **EOF**.

7.1.10 Funktion `fwrite`

Beschreibung:

Schreiben mehrerer Datenobjekte in eine Datei.

Syntax:

```
#include <stdio.h>
size_t fwrite(void *buffer, size_t size,
              size_t number, FILE *file);
```

Bedeutung der Parameter:

`buffer:`	Speicherbereich, der die zu schreibenden Objekte enthält. Die Objekte müssen direkt hintereinander liegen und belegen somit einen Bereich von **`size`***__`number`__ Bytes.
`size:`	Größe eines einzelnen zu schreibenden Datenobjekts
`number:`	Anzahl der zu schreibenden Datenobjekte
`file:`	Zeiger auf gültige **`FILE`**-Struktur

Rückgabewert:

Anzahl der geschriebenen Objekte der Größe **`size`**.

7.1.11 Funktion `fputs`

Beschreibung:

Schreiben einer mit '\0' terminierten Zeichenkette in eine Datei.

Syntax:

```
#include <stdio.h>
int fputs(const char *string, FILE *file);
```

Bedeutung der Parameter:

`file:`	Zeiger auf gültige **`FILE`**-Struktur
`string`	Zeiger auf die zu schreibende Zeichenkette.

Rückgabewert:

Bei fehlerfreier Ausführung: Einen positiven Wert.
Im Fehlerfall: **`EOF`**.

7.1.12 Funktion `fprintf`

Beschreibung:

Formatiertes Schreiben in eine Datei.

Syntax:

```
#include <stdio.h>
int fprintf(FILE *file, const char *format, ...);
```

Bedeutung der Parameter:

`file`	Zeiger auf gültige **`FILE`**-Struktur.
`format`	Format-String (entsprechend **`printf`**).
`...`	Parameter entsprechend dem Format-String.

Rückgabewert:

Anzahl der geschriebenen Zeichen.

7.1.13 Funktion `fseek`

Beschreibung:

Verschieben des Dateizeigers.

Syntax:

```
#include <stdio.h>
int fseek(FILE *file, long offset, int mode);
```

Bedeutung der Parameter:

`file`	Zeiger auf gültige **`FILE`**-Struktur.
`offset`	Offset zum Verschieben des Dateizeigers.
`mode`	Folgende Modi sind vorgesehen: **`'SEEK_SET'`**: Der Offset wird vom Dateianfang aus gerechnet. Der Wert des Offset sollte positiv sein. **`'SEEK_CUR'`**: Der Offset wird von der aktuellen Position des Dateizeigers aus gerechnet. **`'SEEK_END'`**: Der Offset wird vom Dateiende aus gerechnet. Der Wert des Offset sollte negativ sein.

Rückgabewert:

Bei fehlerfreier Ausführung: 0.

Im Fehlerfall: ungleich 0.

7.1.14 Funktion `rewind`

Beschreibung:

Dateizeiger auf den Anfang einer Datei setzen.

Syntax:

```
#include <stdio.h>
void rewind(FILE *file);
```

Bedeutung der Parameter:

`file` Zeiger auf gültige **`FILE`**-Struktur.

7.1.15 Funktion `ftell`

Beschreibung:

Abfrage der Position des Dateizeigers.

Syntax:

```
#include <stdio.h>
long ftell(FILE *file);
```

Bedeutung der Parameter:

`file` Zeiger auf gültige **`FILE`**-Struktur.

Rückgabewert:

Bei fehlerfreier Ausführung: Position des Dateizeigers.
Im Fehlerfall: -1L.

7.2 Beispiele

Beispiel: Einlesen von Messwerten und Ermittlung des Mittelwerts

```
#include <stdio.h>
double mittelwert(FILE *fp);
int main(){
    double wert;
    FILE *fp;
    printf("Mittelwertberechnung, Werte\n");
    printf("zeilenweise eingeben, Ende EOF\n\n");
    fp=fopen("datei.dat", "w");
    while(1==scanf("%lf", &wert)){
        fprintf(fp, "%lf\n", wert);
    }
    fclose(fp);
    /* Mittelwert berechnen und ausgeben */
    fp=fopen("datei.dat", "r");
    printf("Mittelwert : %lf", mittelwert(fp));
    return 0;
}
```

```
double mittelwert(FILE *fp)
{
    double summe=0,wert;
    int n=0;
    while (feof(fp)==0){
        if (fscanf(fp, "%lf", &wert)==1) {
           n++;
           summe+=wert;
        }
    }
   return summe/n;
}
```

Beispiel: Einfaches Kopierprogramm

```
#include <stdio.h>
#define BLOCKWEISE_KOPIEREN 1
int main(int argc, char* argv[]) {
    FILE *in, *out;
    int ch;
#if BLOCKWEISE_KOPIEREN
    char buffer[1000];
    size_t n;
#endif
    if (argc!=3) {
        printf("Verwendung: <Quelle> <Ziel>\n");
        return 1;   /* Ende mit Fehler */
    }
    if ((in = fopen(argv[1], "rb"))==NULL) {
        fprintf(stderr, "Kann %s nicht
                         oeffnen\n", argv[1]);
        return 1;   /* Ende mit Fehler */
    }
    if ((out = fopen(argv[2],"wb"))==NULL) {
        fprintf(stderr, "Kann %s nicht
                         oeffnen\n", argv[2]);
        return 1;   /* Ende mit Fehler */
    }
```

```
    while (!feof(in)) {
#if BLOCKWEISE_KOPIEREN
        n = fread(buffer, 1, 1000, in);
        if (n>0)
            fwrite(buffer, 1, n, out);
#else
    ch=fgetc(in); /* Zeichenweise kopieren */
    if (EOF!=ch) {
        fputc(ch, out);
     }
     else
         break;

#endif
    }  /* of while */
    fclose(in);
    fclose(out);
    return 0;     /* normales Ende */
}  /* of main */
```

7.3 Standardeingabe und Standardausgabe

Tastatur und Bildschirm werden in C auch über **FILE**-Strukturen angesprochen.
Jedes C-Programm hat standardmäßig Zugriff auf die folgenden drei Datenströme vom Typ **FILE ***:

stdin	Standardeingabe (Tastatur)
stdout	Standardausgabe (Bildschirm)
stderr	Standardfehlerausgabe (Bildschirm)

Die Deklaration dieser drei FILE-Strukturen findet sich in der Datei *„stdio.h“*
Diese drei (Pseudo)-Dateien werden vor Aufruf der main-Funktion vom Betriebssystem geöffnet. Ohne Aufruf der **fopen**-Funktion kann somit mit den übrigen Dateifunktionen auf diese drei Datenströme zugegriffen werden.

Beispiel:

Der folgende Aufruf der Ausgabefunktion printf:

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

kann auch wie folgt programmiert werden:

```
fprintf(stdout, "Hallo Welt\n");
```

Beide Programmzeilen sind äquivalent.

8 Zeiger und Felder

8.1 Zeiger und Adressen

Bei einer typischen Maschine kann man sich den Speicherbereich als Bereich von Speicherzellen vorstellen, die fortlaufend nummeriert und adressierbar sind.

Wir haben Zeiger (engl. „pointer“) bereits im Zusammenhang mit Zeichenketten kennengelernt. Zeiger sind Variablen, in der sich die Adresse eines bestimmten Datenobjektes oder einer Funktion befindet. Ihr Speicherbedarf hängt vom Prozessor ab und beträgt meist 2 oder 4 Byte.

Die Verwendung von Zeigern bringt verschiedene Vorteile:

- Speicherersparnis
- Übergabe an Funktionen (call by reference)
- Dynamische Speicherverwaltung
- Rekursive Datenstrukturen
- Effiziente Implementierung von Problemen/Aufgaben (z.B. Sortieren, Zeiger auf Funktionen,...)

Die Programmierung mit Zeigern erfolgt mit Hilfe der unären Operatoren:

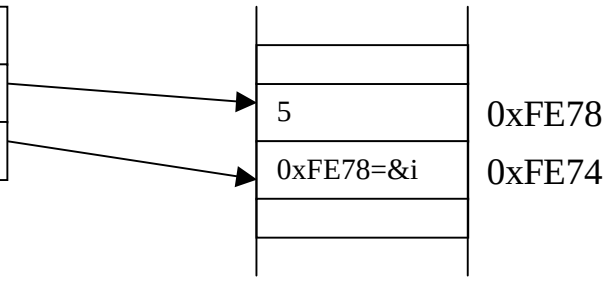
& Adressoperator
***** Inhaltsoperator

Ein Zeiger muss wie jede andere Variable deklariert und definiert werden:

```
int  i;
int  *i_ptr;
i_ptr= &i;
```

Symboltabelle

Bezeichner	Adresse
i	0xFE78
i_ptr	0xFE74



Allgemein:

```
<Typ> *<zeigervariable>;
```

<*Typ*>:	Datentyp des Objektes, auf die der Zeiger verweist.
*****:	Kennzeichnet die Variable als Zeigervariable
<*zeigervariable*>:	Name der Zeigervariablen

Die **Initialisierung** eines Zeigers erfolgt durch die Zuweisung einer Adresse. Ein nicht initialisierter Zeiger enthält undefinierte Bitmuster, die bei Dereferenzierung oftmals zu Programmabstürzen führt!

Beispiele:

```
float x=1, y=2;
float *xf1=&x, *xf2;

*xf2=5.0; /* Fehler, xf2 nicht initialisiert */
xf2=NULL; /* NULL initialisieren (Adresse 0) */
y=*xf1;
*xf1=0;
++*xf1;
(*ip)++;
```

Man beachte: Jeder Zeiger zeigt auf einen festgelegten Datentyp. (Ausnahme: Zeiger auf *void*. Ein *void*-Zeiger kann jeden beliebigen Typ aufnehmen.)

8.2 Zeigerarithmetik

Mit Zeigern sind eingeschränkte Rechenoperationen auf der Grundlage der Speichereinheiten des Basistyps möglich.

Beispiel:

```
int *pa, *pb, n;
```

Vergleich zweier Zeiger	`pa<pb`
Addition eines **int**-Wertes zu einem Zeiger	`pa+n, pa++, ++pa`
Subtraktion eines **int**-Wertes von einem Zeiger	`pa-n`
Subtraktion zweier Zeiger gleichen Typs	`pa-pb`

Bei der Addition und Subtraktion von *int*-Werten zu Zeigern sowie beim Erhöhen und Erniedrigen eines Zeigers durch den **++** und **--** Operator wird der Zeiger um den *int*-Wert multipliziert mit der Länge des zugeordneten Datentyps erhöht. Nachfolgendes Beispiel erläutert diesen Zusammenhang.

Beispiel:

```
short *pa=0x2000;    /* pa auf Adresse 0x2000 */
long  *pb=0x2000;    /* pb auf Adresse 0x2000 */
pa++;       /* pa zeigt nun auf Adresse 0x2002 */
pb++;       /* pb zeigt nun auf Adresse 0x2004 */
pa = pa+3; /* pa zeigt nun auf Adresse 0x2008 */
pb = pb+3; /* pb zeigt nun auf Adresse 0x2010 */
```

Weiterhin kann die Differenz zweier Zeiger gebildet werden, wenn beide Zeiger von gleichem Datentyp sind. Das Ergebnis ist vom Typ Integer und gibt den Abstand zwischen den beiden Zeigern in Einheiten der Objekte des zum Zeiger gehörigen Datentyps an.

Beispiel:

```
char *a=0x1000,*b=0x1008;
long *c=0x1000,*b=0x1008;
int i, j;
i=b-a; /* i wird der Wert 8 zugewiesen */
j=d-c; /* j wird der Wert 2 zugewiesen */
```

Beispiel: Neue Variante von strlen:

```
int strlen(char *s){
   char *p=s;
   while(*p!='\0') p++;
   return p-s;
}
```

Zu beachten ist, dass die unären Operatoren *****, **&**, **++** und **--** von rechts nach links ausgewertet werden:

***pa++**	**pa++** liefert als Ergebnis den bisherigen Wert des Zeigers, der danach um eine Typeinheit erhöht wird. ***pa++** dereferenziert also das Objekt, auf das **pa** vor dem Erhöhen zeigt.
(*pa)++	Erst wird der *****-Operator auf **pa** angewendet, dann der **++**-Operator auf das dereferenzierte Objekt. Der Zeiger bleibt unverändert. Man greift also auf das Objekt zu, auf das **pa** gerade zeigt und erhöht dessen Wert um 1.
***++pa**	Der Wert von **pa** wird erst erhöht. Dann wird auf das Objekt zugegriffen, auf welches der erhöhte Zeiger **pa** zeigt.
++*pa	Das Objekt, auf das **pa** zeigt, wird erst um 1 erhöht und dann verwendet.

Weitere Operationen (z.B. Addition und Multiplikation von Zeigern) sind verboten.

Eine Besonderheit ist der **void-Zeiger**. Er enthält die Adresse einer Speicherzelle ohne Festlegung des Datentyps.

Beispiel:

```
void *x, *y;
double d;
long int l;
x=&d;
y=&l;
```

Beliebige Zeiger können ohne Verwendung des cast-Operators in **void**-Zeiger umgewandelt und wieder zurückgewandelt werden (sonst ist immer der cast-Operator erforderlich). Erhöhung und Erniedrigen eines **void**-Zeigers erfolgt in 1 Byte-Schritten (wie ein Zeiger auf **char**).

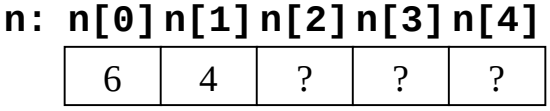
8.3 Zusammenhang zwischen Feldern und Zeigern

Gleichartige, logische zusammenhängende Objekte können zu Feldern (Array, Vektor) zusammengefasst werden (siehe Abschnitt 4).

Beispiel (siehe Abschnitt 4):

```
int n[5];
n[0]=6;
n[1]=4;
```

Es wird ein Feld namens **n** mit 5 *integer*-Zahlen vereinbart, d.h. es wird ein zusammenhängender Speicherplatz für 5 *integer*-Zahlen reserviert.



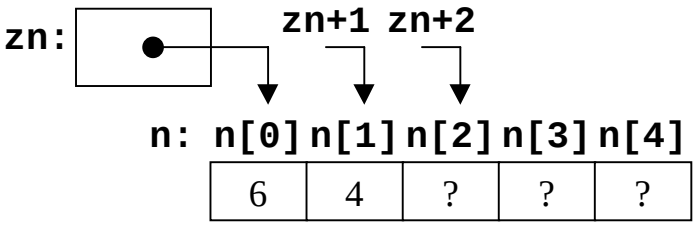
Bemerkungen:

- Das erste Feldelement eines Feldes aus N Elementen hat den Index 0, das letzte Element hat den Index N-1.
- In C werden mögliche Bereichüberschreitungen nicht geprüft. Mit:
  `n[5]=7;`
  wird beispielsweise der Speicherplatz hinter **n[4]** (siehe oberes Beispiel) zerstört (→ Laufzeitfehler).
- Der Name des Feldes ohne nachfolgende Indizierung **[**...**]** hat die Bedeutung einer Adresse. Der Name stellt jedoch einen konstanten Zeiger und keine Zeigervariable dar. Ist **n** der Name eines Feldes sind Ausdrücke wie **n++** und **n=zn** (Mit **zn** als Zeigervariable) somit nicht erlaubt.
- **n** entspricht **&n[0]**.
- **n+i** entspricht **&n[i]**.
- **n[i]** entspricht ***(n+i)**.
- Die (absolute) Adresse des i-ten Feldelementes **n[i]** aus obigem Beispiel lässt sich berechnen zu:
  *Adresswert(n) + i*sizeof(int).*
  Dies ist jedoch kein C-Code!!

Man kann auch mit Hilfe von Zeigervariablen auf die Elemente eines Feldes zugreifen.

Beispiel:

```
int n[5];
int *zn;
zn=n;          /* oder zn=&n[0] */
*zn     = 6;
*(zn+1) = 4;
```



Der Zeigerwert **zn** zeigt auf das erste Element **n[0]**, **zn+1** zeigt auf das zweite Element **n[1]** usw. Somit entspricht ***(zn)** dem Feldzugriff **n[0]** und ***(zn+1)** entspricht **n[1]** usw.

Wichtig: Feldname und Zeigervariable sind zu unterscheiden. Der Zeiger ist eine Variable mit zugewiesenem Speicherplatz. Ausdrücke wie **zn=n** und **zn++** sind erlaubt.

Ein Feldname stellt in C einen konstanten Zeiger dar. Zum Zugriff auf ein Feldelement wird beim **[]**-Operator der Feldname und der Index des gewünschten Elements angegeben (z.B. **n[3]**). Der []-Operator fordert jedoch keineswegs einen konstanten Zeiger, sondern es kann ein beliebiger Zeiger angegeben werden. Somit kann z.B. bei gegebener Zeigervariablen **zn** statt mit ***(zn+3)** auch mittels **zn[3]** auf das dritte Feldelement zugegriffen werden.

Beispiel:

```
int main(){
   int n[10], *zn;
   zn=n;   /* Wichtig: Zeiger initialisieren */
   zn[0]=6;
   zn[1]=7;
   printf("Ergebnis %i %i\n", n[0], n[1]);
   return 0;
}
```

Wichtig ist jedoch, dass die Zeigervariable vorher korrekt initialisiert wurde und wirklich auf ein Feld im Speicher schreibt. Ansonsten erfolgt der Feldzugriff auf eine zufällige Stelle im Speicher (→ Laufzeitfehler).

8.4 Felder als Funktionsparameter

Bei der Übergabe von Feldern an Funktionen gelten die folgenden Bemerkungen:

- Von einem Feld wird die Adresse als Parameter an eine Funktion übergeben.
- Zur Deklaration des zu einem Feld zugehörigen Funktionsparameters sind zwei Formen äquivalent: Eine Zeigerdeklaration oder eine Felddeklaration ohne spezifizierte Größe.
  Beispiel: „**`double *v`**“ und „**`double v[]`**“ sind äquivalent.
- Die Dimension von Feldern wird nicht implizit mit übergeben. Wird die Dimension eines Feldes in der Funktion benötigt, muss sie über einen zusätzlichen Parameter separat übergeben werden.

Programmbeispiel zur Übergabe eines Feldes an eine Funktion:

```
#define MAXDIM 10

double skalarprodukt(double *, double *, int);

int main() {
    double x[MAXDIM], y[MAXDIM], produkt;
    int result, n=0;
    while(n<MAXDIM && EOF!=scanf("%lf %lf", &x[n],&y[n])) {
        ++n;
    }
    produkt=skalarprodukt(x, y, n);
    printf("%lf", produkt);
    return 0;
}

double skalarprodukt(double *v, double *w,
                     int dim) {
   double skprod=0;
   int i;
   for(i=0; i<dim; i++) {
       skprod+=v[i]*w[i];
   }
   return (skprod);
}
```

Beispiel: Stack mit den Funktionen *pop*, *push* und *stapel_aus*

```
#include <stdio.h>
#define MAX 10

int stapel[MAX], top=0;  /* globale Variable */
int push(int wert);
int pop(void);
int stapel_aus(void);

int main() {
    int zahl;
    char ch;
    printf("Stapel, Auswahl durch grossen Buchstaben
            des Befehls:\n\n");
    while (1) {
        printf("pUsh, pOp, Ausgabe, Exit: ");
        scanf("%c", &ch);
        if (ch=='\n') {
            scanf("%c", &ch);
        }
        if(ch=='u' || ch=='U') {
            printf("Bitte Zahl eingeben     : ");
            scanf("%d", &zahl);
            push(zahl);
         }
        else if(ch=='o' || ch=='O') {
            printf("oberstes Element: %d\n", pop());
        }
        else if (ch=='a' || ch=='A') {
            stapel_aus();
        }
        else if(ch== 'e' || ch=='E') {
            return 0;
        }
        else {
            printf("\nFehler: Falsche Eingabe\n");
        }
        return 0;
    }  /* of while */
} /* of main
```

```
int push(int wert){
   if(top==MAX) {
      printf("Stapelueberlauf! ");
      return 1;
   }
   else {
      stapel[top++]=wert;
      return 0;
   }
}
int pop(void){
   if(top == 0) {
      puts("Stapel ist leer");
      return 2;
   }
   else {
      return stapel[--top];
   }
}
int stapel_aus(void) {
   int zaehl;
   if (top==0) {
      puts("Stapel ist leer\n");
      return 1;
   }
   else {
      printf("\nInhalt des Stapels:    \n");
      for(zaehl=top-1; zaehl>=0; zaehl--){
         printf("              %d\n", stapel[zaehl]);
      }
      printf("\n");
      return 0;
   }  /* of if
}   /* of stapel_aus */
```

8.5 Mehrdimensionale Felder

Mehrdimensionale Felder sind in C als Felder von Feldern definiert.

Syntax (2-dimensionales Feld):

Definition (*n*, *m*: int-Werte):

```
Speicherklasse Typ Name [n][m];
```

Aufruf (Zeilenindex *i*: 0≤*i*<*n*; Spaltenindex *j*: 0≤*j*<*m*):

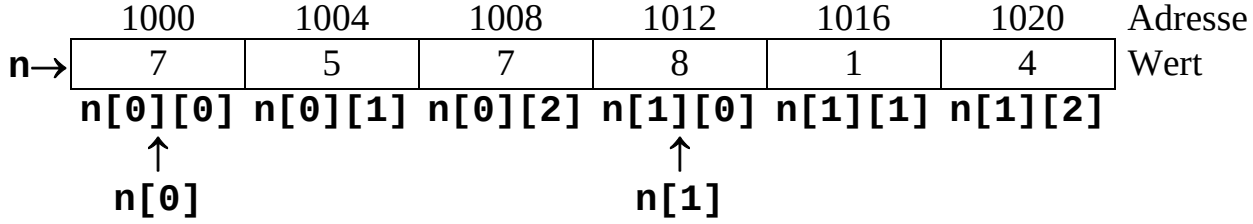
```
Name[i][j]
```

Beispiel:

```
int n[2][3];
n[0][0]=7;
n[0][1]=5;
n[0][2]=7;
n[1][0]=8;
n[1][1]=1;
n[1][2]=4;
```

$$n = \begin{pmatrix} 7 & 5 & 7 \\ 8 & 1 & 4 \end{pmatrix}$$

Die Elemente des mehrdimensionalen Feldes werden, beginnend mit der niedrigsten, linksstehenden Dimension zeilenweise gespeichert:



Bedeutung des Namens **n** bei mehrdimensionalen Feldern

Der Feldname **n** in obigem Beispiel steht (wie in Abschnitt 8.3 erläutert) für die Adresse des ersten Feldelements. Da **n** im zweidimensionalen Fall ein Feld von Feldern ist, zeigt dieser konstante Zeiger **n** im Beispiel auf ein Feld von 3 Elementen vom *integer*-Typ. Die Variable **n** ist daher vom Typ „Zeiger auf Feld von 3 Elementen“, welcher wie folgt in C beschrieben werden kann:

```
int *[3]
```

oder

```
int [][3].
```

Der Ausdruck ***n** liefert einen Zeiger auf das Feld **n[0]**, der Ausdruck ***(n+1)** zeigt entsprechend auf das Feld **n[1]**. Allgemein referenziert der Ausdruck **n[i]** das i+1-te Feldelement von **n**. Im Beispiel referenziert der Ausdruck **n[i]** ein eindimensionales Feld aus *integer*-Elementen. Er steht somit für einen konstanten Zeiger auf Elemente des Typs *integer*, der auf die Adresse des ersten Elements **n[i][0]** zeigt. Auf die skalare Komponente **n[i][j]** des zweidimensionalen Feldes kann somit alternativ wie folgt zugegriffen werden:

n[i][j]	Feldelement **j** von Feld (Feldelement **i** von Feld **n**)
***(*(n+i)+j)**	(Inhalt des Elements mit Abstand **j** von Zeiger (Inhalt des Elements mit Abstand **i** von Zeiger **n**))
(*(n+i))[j])	Feldelement **j** von Feld (Inhalt des Elements mit Abstand **i** von Zeiger **n**)
***(n[i]+j)**	(Inhalt des Elements mit Abstand **j** von Zeiger (Feldelement **i** von Feld **n**))

Der Bezeichner **n** und der Ausdruck **n[0]** repräsentieren beide konstante Zeiger, welche auf die gleiche physikalische Adresse zeigen. Sie sind jedoch keineswegs äquivalent, da sie unterschiedlichen Typen zugeordnet sind:

Typ von **n**: **int [][3]** oder **int *[3]**
Typ von **n[i]**: **int [3]**

Der erste Zeiger **n** zeigt im Beispiel auf die Adresse des ersten Feldes **n[0]** aus *integer*-Elementen. Dieses Feld beginnt im Speicher an der Adresse seines ersten Elementes **n[0][0]**. Der zweite Zeiger **n[0]** zeigt auf die Adresse seines ersten Elements **n[0][0]**. Somit wird beiden Zeigern der gleiche Adresswert zugeordnet, obwohl sie von unterschiedlichem Typ sind.
Der unterschiedliche Typ kommt jedoch zum Tragen, wenn zum Zeigerwert z.B. der Wert 1 hinzu addiert wird (Zugriff auf das zweite Element).

Beispiel:

Annahme: **n** und **n[0]** zeigen auf Adresse *1000*.
n+1 entspricht dann der Adresse *1012*
n[0]+1 entspricht dann der Adresse *1004*.

Beispiel: Initialisierung eines Feldes und Zugriff über verschiedene Ausdrücke

```
#include <stdio.h>

int main() {
   double a[2][3]={{.1,12.0,1.3},{2.1,2.2,2.3}};
   printf("%f=%f=%f=%f\n", a[1][1], (*(a+1))[1],
          *(a[1]+1),*(*(a+1)+1));
}
```

Beispiel: Transponieren einer Matrix

```
#define MAXDIM 100 /* Anzahl Zeilen/Spalten */

void transpo(int *, int *, double [][MAXDIM]);

int main() {
   int zeilen, spalten;
   double A[MAXDIM][MAXDIM]);
   ...
   transpo(&zeilen, &spalten, A);
   ...
}
```

```
void transpo(int *n, int *m, double A[][MAXDIM]){
                    /* oder double (*A)[MAXDIM] */
   int i, j, max=(*m>*n)? *m : *n;
   double hilf;
   for (i=0; i<max; i++){
      for (j=0; j<i; j++) {
         hilf=A[i][j];
         A[i][j]=A[j][i]; /* Transponieren */
         A[j][i]=hilf;
      }
   }
   i=*n;
   *n=*m; /* Austausch der Dimensionen */
   *m=i;
}
```

8.6 Felder von Zeigern

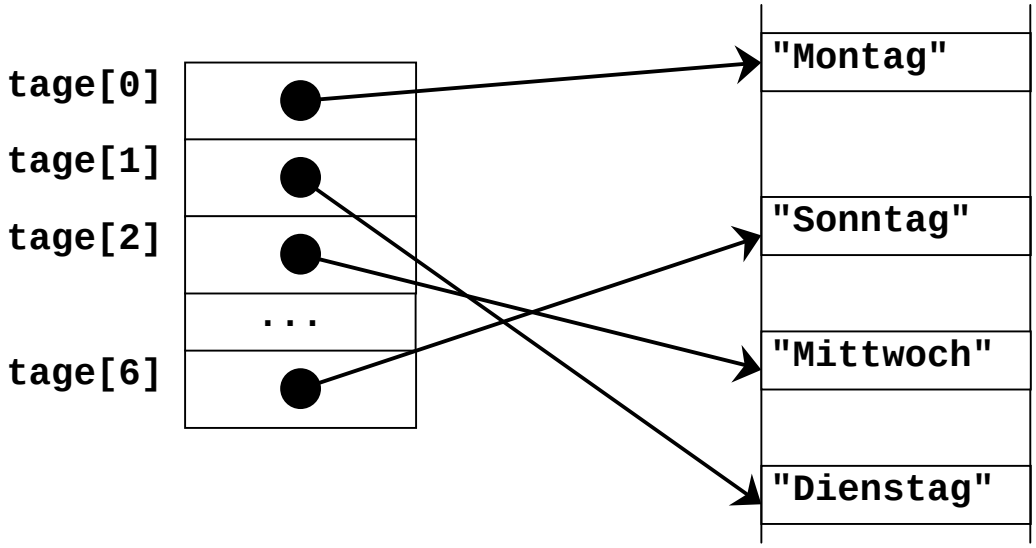
In C können auch Felder von Zeigern definiert werden. Felder von Zeigern werden in C sehr häufig verwendet. Ein solches Feld enthält nur Adressen. Die Adressen müssen alle vom selben Typ sein.
Die Definition:

```
short *aptr[10];
```

ist beispielsweise die Definition eines Feldes, das 10 Zeiger vom Typ „Zeiger auf *short*-Variable“ enthält. Jedes Element dieses Feldes (**`aptr[0]`**, ..., **`aptr[9]`**) ist vom Typ **`short *`**.
Felder von Zeigern werden häufig im Zusammenhang mit Zeichenketten verwendet (siehe Abschnitt 4). Dort hat jedes Feldelement den Typ „Zeiger auf *character*-Variable“ (bzw. **`char*`**) und kann auf das erste Element einer Zeichenkette verweisen.

Beispiel: Zeigerfeld für Zeichenketten

```
char *tage[7];
tage[0]="Montag";
tage[1]="Dienstag";
tage[2]="Mittwoch";
tage[3]="Donnerstag";
tage[4]="Freitag";
tage[5]="Samstag";
tage[6]="Sonntag";
```



Der Zugriff auf die Texte geschieht, wie für Felder fester Größe oben erläutert, mittels Indizierung oder mittels Zeigerarithmetik.

```
printf("%s\n", tage[0]);
printf("%s\n", *(tage+1));
printf("%c", *tage[0]);
printf("%c", *(tage[3]+1));
printf("%c", *(*(tage+1)+3));
printf("%c", tage[3][0]);
```

Ausgabe der **`printf`**-Anweisungen:

```
Montag
Dienstag
M
Mo
Mon
MonD
↑(Hier steht der Cursor)
```

Ein Feld von *character*-Zeigern auf Zeichenketten ist ein spezieller Fall des Feldes von Zeigern, die auf die Anfangselemente von Feldern verweisen. Diese Art der Verwaltung von Feldern (z.B. Zeichenketten) bietet einige Vorteile bei der Verarbeitung von Feldern:

- Felder unterschiedlicher und beliebiger Länge können über das Zeigerfeld zusammengefasst werden.
- Beim Sortieren der Felder ist nur ein Vertauschen von Zeigerinhalten (Adressen) im Zeigerfeld erforderlich.
- Es können mehrere Zeigerfelder definiert werden, welche auf die gleichen Felder

zeigen, diese aber nach unterschiedlichen Kriterien sortieren.

Nicht zu verwechseln ist ein Feld von Zeigern mit mehrdimensionalen Feldern:

```
char  a[10][20]; /* Speicherung von 200 char-Werten */
char *a[10];     /* Speicherung von 10 Zeigern auf char, Texte
                    können unterschiedlich lang sein */
```

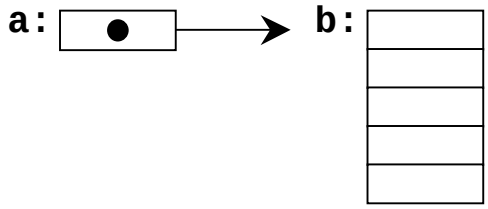
In beiden Fällen wird auf das Element mit Index (2,3) mit **a[2][3]** zugegriffen.

Zeiger auf Felder:

In Ergänzung zum Feldnamen, der einen konstanten Zeiger repräsentiert, kann auch ein variabler Zeiger auf ein Feld vereinbart werden.

Beispiel:

```
long int (*a)[5]; /* Zeiger auf ein Feld
                     mit 5 Elementen.
                     Achtung: Es wird kein
                     Feld definiert */
long int b[5];    /* Feld mit 5 Elementen */
...
a=&b; /* Zuweisung der Feldadresse zum Zeiger*/
b[1]=(*a)[0]; /* (Element 1) = (Element 0) */
```



Zum Abschluss soll nochmals auf den Unterschied zwischen einem Feld von Zeigern:

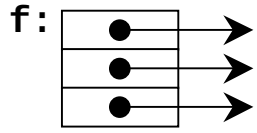
```
long int *f[3];   /* ein Feld von Zeigern */
```

und einem Zeiger auf ein Feld:

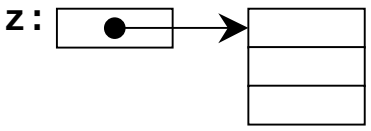
```
long int (*z)[3]; /* Zeiger auf ein Feld  */
```

hingewiesen werden. Nachfolgendes Diagramm verdeutlicht den Unterschied:

Feld von Zeigern



Zeiger auf Feld



8.7 Zeiger auf Funktionen

In C gibt es neben Zeiger auf Datenobjekte auch Zeiger auf Funktionen.
Ein Zeiger auf eine Funktion hat als Wert die Anfangsadresse des Unterprogramms in Maschinensprache, das die Funktion ausführt.

Beispiel:

Die Definition:

```
int (*f) (double);
```

bezeichnet einen Zeiger mit Bezeichner **f** auf eine Funktion, die einen **double**-Wert als Argument verlangt und einen **int**-Wert als Rückgabewert zurückliefert.
Die Klammerung **(*f)** ist notwendig, da der **()**-Operator stärker als der *****-Operator bindet. Die sonst entstehende Definition **int *f(double)** bezeichnet eine Funktion, die einen *integer*-Zeiger als Rückgabewert liefert.

Zeiger auf Funktionen werden dort angewendet, wo Funktionen als Eingabeparameter erforderlich sind (z.B. bei numerischen Routinen wie Integralberechnung, Differentialgleichungs-Lösern,...) .

Der Name einer C-Funktion hat die Bedeutung eines konstanten Zeigers auf die Funktion und gibt somit die Startadresse der Funktion an.

Beispiel:

In der folgenden Definition:

```
double g(double x) {...}
```

steht der Name g für einen konstanten Zeiger auf die Startadresse der Funktion.

Beispiel: Numerische Differentiation:

$$f'(x) \approx \frac{f(x+h) - f(x-h)}{2h}$$

Die Funktion mit Namen **diff** berechnet die Näherungswerte für die Ableitung einer beliebigen, differenzierbaren Funktion **f** an einer Stelle **x**. Für die Berechnung benötigt die Funktion **diff** die folgenden Eingaben:

- Startwert *x*.
- Eine Funktion *f(x)*.
- Der Wert h wird fest vorgegeben.

Der Rückgabewert der Funktion ist die Näherung für *f'(x)*.

```
#include <math.h>
#include <stdio.h>
#define h  0.001

double test(double);

double diff(double (*)(), double);
```

```
int main(){
   double a;
   scanf("%lf", &a);
   printf("Ableitung von f ist %lf", diff(test, a));
}
double test(double x){
   return cos(2*x*x);
}
double diff( double (*f)(double), double x){
   double ergebnis;
   ergebnis=((*f)(x+h)-(*f)(x-h))/(2*h);
   return ergebnis;
}
```

8.8 Argumente aus der Kommandozeile

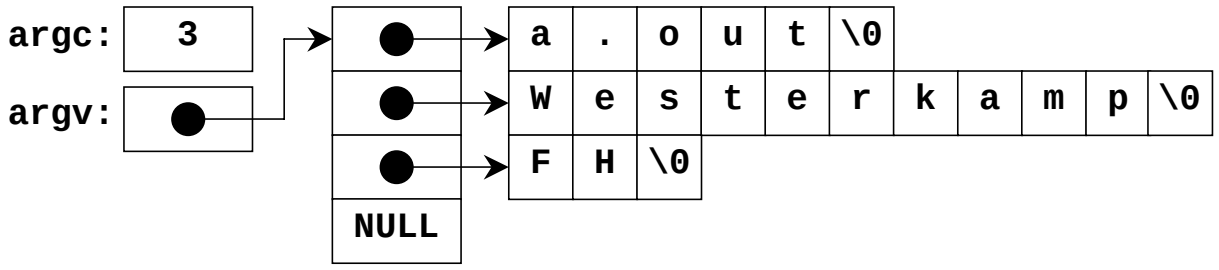
Einem ausführbaren C-Programm können (so wie den meisten Kommandos unter Unix oder DOS) beim Aufruf durch Leerzeichen getrennte Argumente über die Kommandozeile übergeben werden. Diese Argumente werden vom Betriebssystem gelesen und der **`main`**-Funktion beim Aufruf als Parameter übergeben.
Die **`main`**-Funktion muss dann wie folgt deklariert werden:

```
int main(int argc, char *argv[]);
```

Der erste Parameter **`argc`** enthält die Anzahl der Argumente. Der zweite Parameter **`argv`** ist ein Zeiger auf ein Feld von Zeigern auf Zeichenketten. Die Größe des Feldes ist durch den Wert des ersten Parameters **`argc`** spezifiziert. Jeder Zeiger des Feldes zeigt auf eine Zeichenkette, welche als Argument beim Aufruf des Programms auf der Kommandozeile spezifiziert wurde.

Beispiel:

Beim Aufruf des Programms *a.out* mit den Argumenten ‘’Westerkamp’’ und ‘’FH’’ ergibt sich folgende Struktur für *argv*:



Im Beispiel ist dargestellt, dass der Zeiger Feld, auf den **`argv`** zeigt, um ein Element größer ist als der in **`argc`** spezifizierte Wert. Das letzte Argument im Zeigerfeld (also **`argv[argc]`**) enthält immer den NULL-Zeiger.

Beispiel: Echo Kommando

Folgendes Beispiel simuliert das Echo-Kommando. Es druckt alle Argumente inklusive des Kommandonamens aus.

```
#include <stdio.h>
int main(int argc, char *argv[]){
    int i;
    for(i=0; i<argc; ++i) {
        printf("%s"', argv[i]);
    }
    printf("\n");
}
```

Da das letzte Element des Zeigerfeldes immer den Wert NULL enthält, kann das obige Progamm auch als nachfolgende Zeigervariante realisiert werden.

```
#include <stdio.h>
int main(int argc, char *argv[]){
    do {
       printf("%s ", *argv);
    }
    while(NULL != *++argv);
    printf("\n");
    return 0;
}
```

Die erste Zeichenkette, auf die der Zeiger **`argv[0]`** zeigt, enthält den Kommandonamen, mit dem das Programm gestartet wurde. Unter UNIX entspricht er dem Namen der Datei, die den ausführbaren Programmcode enthält (z. B. a.out).
Der Kommandoname wird bei der Anzahl der Argumente in **`argc`** mitgezählt.

An ein C-Programm können nur Zeichenketten übergeben werden. Will man eine Zahl übergeben, muss die zugehörige Zeichenkette in eine Zahl gewandelt werden. Dazu dienen die Funktionen:

`double atof(char *)` oder **`sscanf (char *, char *,...).`**

Beispiel:

Das nächste Programm erwartet eine Dezimalzahl als Kommandozeilen-Argument. Die Zeichenkette wird in das *double*-Format umgewandelt.

```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
int main(int arg c, char *argv[]) {
    double y;
    if (argv[1])  {
        y=atof(argv[1]);
        printf("\nsin(%lf)=%lf, cos(%lf)=%lf\n",
```

```
                    y, sin(y), y, cos(y));
        } else printf("\n Argument fehlt\n");
    }
```

8.9 Beispiele für Typvereinbarungen

`int a;`	*integer*-Variable
`double *a;`	Zeiger auf *double*
`float mat[10];`	Feld mit 10 Komponenten vom Typ *float*
`double Zahl[10][20];`	2-dimensionales Feld mit 10 Zeilen und 20 Spalten
`long int (*FD)[100];`	Zeiger auf ein Feld mit 100 Komponenten
`char *Zeilen[100];`	Feld von 100 Zeigern auf *character*
`short *g(int b);`	Funktion, die einen *integer*-Wert als Parameter fordert und einen Zeiger auf *short* zurückgibt.
`double (*g)` `(float x1,float x2);`	**g** ist ein Zeiger auf eine Funktion, die als Ergebnis einen Zeiger auf *double* liefert und als Argumente zwei *float*-Werte erwartet.
`int **a;`	Zeiger auf Zeiger auf *integer*-Wert
`long **f()`	**f** ist eine Funktion mit Rückgabe Zeiger auf Zeiger auf *long*-Wert.
`long *(*f)()`	**f** ist ein Zeiger auf eine Funktion mit Rückgabewert Zeiger auf *long integer*.
`long (**f)()`	**f** ist ein Zeiger auf einen Zeiger auf eine Funktion mit Rückgabewert *long integer*.
`long **(**f)()`	**f** ist ein Zeiger auf einen Zeiger auf eine Funktion mit Rückgabewert Zeiger auf Zeiger auf *long*.
`int(*f(int(*)()))();`	Funktion **f**, die als Argument einen Zeiger auf eine Funktion mit Rückgabewert *integer* erwartet und einen Zeiger auf eine ebensolche Funktion zurückliefert.

Bemerkung:

Es gibt in C

- keine Felder, deren Komponenten Funktionen sind und
- keine Funktionen, die ein Feld als Rückgabewert haben.

Es gibt aber

- Felder, deren Komponenten Zeiger auf Funktionen sind und
- Funktionen, die einen Zeiger auf ein Feld zurückgeben.

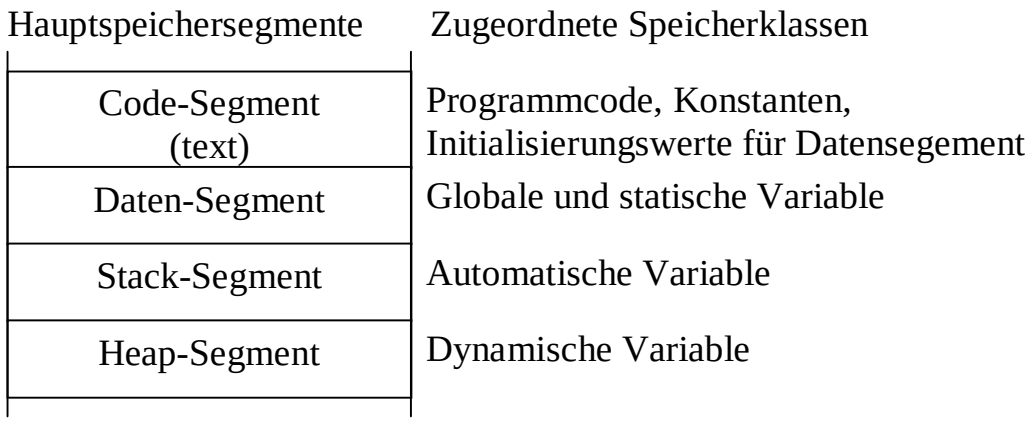
9 Speicherklassen, Attribute und Gültigkeitsbereiche

9.1 Speicherklassen

Ein Programm kann Variablen mit unterschiedlichen Eigenschaften enthalten.

- **Automatische Variable** (transiente Variable) werden lokal angelegt. Nach Abarbeitung eines zugehörigen Programmteils wird der zugeordnete Speicher wieder freigegeben.
- **Globale und statische Variable** (permanente Variable) werden beim Starten des Programmes angelegt, der zugeordnete Speicher bleibt während der gesamten Programmlaufzeit der Variablen zugeordnet.
- **Dynamische Variable** werden während der Laufzeit durch den Programmcode angelegt und auch wieder freigegeben. Ihnen wird kein Variablenname zugeordnet, der Zugriff erfolgt über (automatische, statische oder globale) Zeigervariable.
- Schließlich besitzt ein Programm noch **konstante Werte**, die durch den Programmcode nicht verändert werden können.

Diese verschiedenen Speicherklassen der Variablen, auch als Variablenklassen bezeichnet, werden auf unterschiedliche Speicherbereiche (Speichersegmente) eines Programms abgebildet. Nachfolgende Abbildung zeigt diesen Zusammenhang:



Beim Start eines Programmes wird zunächst vom Betriebssystem das Code-Segment (häufig auch als text-Segment bezeichnet) geladen und der Speicherplatz für die anderen Segmente bereitgestellt. Dann wird vom Betriebssystem der Programmcode im Code-Segment angesprungen. Durch Routinen, die mit dem Compiler mitgeliefert und beim Binden zu den Anwenderfunktionen hinzugefügt werden, wird zunächst das Datensegment initialisiert. Diese Routinen rufen anschließend die **`main`**-Funktion auf und starten damit den vom Programmierer in der Sprache C spezifizierten Code.

9.1.1 Automatische Variable

Alle bisher betrachteten Variablen wurden im Funktionsrumpfes zu Beginn im Vereinbarungsteil definiert. Diese Variablen sind lokal innerhalb des gesamten Funktionsrumpfes gültig.
Möchte man Variablen noch detaillierter einem eingeschränkten, lokalen Code-Bereich zuordnen, kann man auch am Beginn jeder Verbundanweisung (siehe Abschnitt 4) einen Vereinbarungsteil vorsehen.
Beide Variablendefinitionen erzeugen **automatische** Variable. Eine automatische Variable wird beim Aufruf einer Funktion bzw. beim Eintritt des Programmes in eine Verbundanweisung jedesmal neu auf dem Stack (siehe Abschnitt 5) angelegt. Nach dem Anlegen ist der Wert der Variable undefiniert. Der Wert der Variablen aus einem früheren Funktionsaufruf oder einem früheren Durchlauf der Verbundanweisung ist nicht mehr vorhanden. Vor einer sinnvollen Verwendung muss daher der Variablen ein Wert zugewiesen werden. Diese Wertzuweisung kann innerhalb der Definition oder später in einer separaten Zuweisung erfolgen.

Beispiel:

```
#include <stdio.h>

int main(){
   int i;
   printf("Wert von i: %d\n", i);
}
```

Der Wert von **i** ist zufällig!!

Beispiel:

```
#include <stdio.h>

int test(int v);

int main(){
   printf("Wert1 test: %d\n", test(1));
   printf("Wert2 test: %d\n", test(1));
}

int test(int v){
   int i = 0; /* i lokal in Funktion test */
   i = i+v;
   {  int j;  /* j lokal in Verbundanweisung */
      for (j=0; j<10; j++) {
         i = i+j;
      }
   }
   return i;
}
```

Das Beispiel zeigt die Verwendung einer lokalen, automatischen Variablen (Variable **j**), die zu Beginn eines Blocks definiert wird. Die Variable **i** wird bei jedem Aufruf der Funktion Test neu auf den Wert 0 initialisiert. Somit führt der zweimalige Aufruf der Funktion **test** jedes Mal zum gleichen Ergebnis und die beiden obigen **printf**-Anweisungen geben den gleichen Wert aus. (Übung: Welches Ergebnis liefert die Funktion **test**?)

Da die automatischen Variablen auf dem Stack angelegt werden, sind ihre Adressen nach dem Übersetzen und Binden des Programmes nicht bekannt. Sie ergeben sich erst beim Start der Funktion zur Laufzeit. Auch variieren die Speicherpositionen der Variablen auf dem Stack, je nachdem von wo aus eine Funktion aufgerufen wird.
Automatische Variable können durch Voranstellen des Schlüsselwortes **auto** als automatische Variable gekennzeichnet werden.
Die C-Syntax erlaubt eine zweite Kennzeichnung für häufig verwendete automatische Variable, die zur Optimierung des Laufzeitverhaltens in Registern der CPU gehalten werden sollten. Diese Variable werden durch Voranstellen des Schlüsselwortes **register** gekennzeichnet. Es wird jedoch nicht garantiert, dass solche Variable wirklich in Register gelegt werden, sie können vom Compiler auch auf dem Stack abgelegt werden. Die Kennzeichnung dient lediglich als Hinweis für den Optimierungslauf des Compilers.

Beispiel:

```
long a, b;          /* Automatische Variable,
                       auch ohne Kennzeichnung */
auto double f, g;   /* Explizite Kennzeichnung
                       als automatische Variable */
register int i, j; /* Automatische Register-
                       Variable */
```

9.1.2 Statische, lokale Variable

Statische, lokale Variable werden, wie auch die automatischen Variablen, im Vereinbarungsteil am Beginn eines Funktionsrumpfes oder einer Verbundanweisung definiert. Sie werden durch das Schlüsselwort **static** gekennzeichnet.
Statische, lokale Variable werden im Datensegment angelegt, daher ist jeder dieser Variablen immer eine eindeutige Speicherzelle und damit eine eindeutige Adresse zugeordnet. Ein zugewiesener Wert bleibt erhalten, bis ein neuer Wert zugewiesen wird.
Einer lokalen, statischen Variablen kann bei der Definition ein Initialisierungswert zugewiesen werden. Der Initialisierungswert wird der Variablen vor dem Start der **main**-Funktion einmal zugewiesen. Wird kein Initialisierungswert spezifiziert, wird eine statische, lokale Variable zu 0 initialisiert.

Beispiel:

```
#include <stdio.h>
int test();

int main(){
    printf("test: %d\n", test());
    printf("test: %d\n", test());
    printf("test: %d\n", test());
    return 0;
}
int test(){
    static int i=0;
    return i++;
} /* i ueberlebt das Ende der Verbundanweisung */
```

Das Programm ergibt folgende Ausgabe:

```
test: 0
test: 1
test: 2
```

Rekursive Funktionen: Bei rekursiven Funktionen ist zu beachten, dass für jede rekursive Instanz einen eigener Satz automatischer Variablen auf dem Stack angelegt wird, alle Instanzen jedoch auf die gleichen lokalen, statischen Variablen zugreifen.

Beispiel:

```
#include <stdio.h>

void test(int n);

int main(){
   test(3);
   return 0;
}

void test(int n){
   int i=0;
   static int j=0;
   if (n) {
      test(n-1);
   }
   printf("n, i, j: %d, %d, %d\n", n, i++, j++);
}
```

Das Programm führt zu folgender Ausgabe auf dem Bildschirm:

```
n, i, j: 0, 0, 0
n, i, j: 1, 0, 1
n, i, j: 2, 0, 2
```

```
n, i, j: 3, 0, 3
```

9.1.3 Globale Variable

Globale Variable werden beim Programmstart außerhalb von Funktionen eingerichtet und bleiben bis zum Ende des Programms verfügbar. Aus unterschiedlichen Funktionen kann direkt auf die globale Variable zugegriffen werden, ohne diese mittels Zeiger an die Funktionen zu übergeben. In C gibt es für globale Variable eine Sichtbarkeit innerhalb einer Datei und eine Sichtbarkeit im gesamten Programm.

Globale Variable mit Sichtbarkeit im gesamten Programm

Die Vereinbarung einer Variablen außerhalb einer Funktion ohne Schlüsselwort für die Speicherklasse definiert eine globale Variable, die im gesamten Programm sichtbar ist. Da der Code eines Programmes über mehrere Dateien aufgeteilt werden kann, eine Variable jedoch nur einmal definiert werden darf, muss eine Verbindung der globalen Variablen zu den übrigen Dateien hergestellt werden. Dazu dient das Schlüsselwort **`extern`**. Es deklariert die Variable, führt aber nicht zur Zuweisung von Speicher.

Globale Variable mit Sichtbarkeit innerhalb einer Datei

Die Vereinbarung einer Variablen außerhalb einer Funktion mit Schlüsselwort **`static`** definiert eine globale Variable, die innerhalb der Datei sichtbar ist.

Globale Variable werden in C grundsätzlich initialisiert. Werden keine Initialisierungswerte angegeben, wird eine Initialisierung mit 0 durchgeführt.

Beispiel:

Datei main.c:

```
#include <stdio.h>
void test(int n);
int g;

int main(){
   printf("1: g=%d\n", g);
   test(3);
   printf("2: g=%d\n", g);
   test(7);
   printf("3: g=%d\n", g);
   test(0);
   printf("4: g=%d\n", g);
   return 0;
}
```

Datei test.c

```
extern int g;
static int d

void test(int n){
   g=d;
   d=n;
}
```

9.1.4 Dynamische Variable

Dynamische Variable werden durch Programmcode angefordert. Sie besitzen keinen Variablennamen, sondern werden nur über zugeordnete Zeiger angesprochen. Die Lebensdauer ist somit ebenfalls durch den Programmcode bestimmt. Eine Initialisierung dynamischer erfolgt nicht bei der Anforderung, sondern muss explizit durch den Programmcode erfolgen.
Eine genaue Betrachtung dynamischer Datenobjekte erfolgt in Abschnitt 10.

9.1.5 Konstante Variable

ANSI-C erlaubt die Spezifikation von konstanten Variablen. Sie werden wie Variablen vereinbart, der Definition wird jedoch das Schlüsselwort **`const`** vorangestellt. Damit erhält die Variable ein Attribut, dass sie nicht geändert werden soll. Der Compiler verhindert dann das Ändern des Wertes (oder warnt zumindest davor). Die Speicherklasse ist äquivalent zur Speicherklasse, die sich auch ohne das Attribut ergeben würde. Somit gelten die bisherigen Ausführungen zu lokalen und globalen Variablen auch für konstante Variable.

Beispiel:

```
const int Anzahl = 100;
const double pi =3.1415927;
```

Eine konstante Variable muss bei der Definition initialisiert werden, da eine spätere Wertzuweisung ja durch das **`const`**-Attribut verhindert werden soll.

Bei der Definition konstanter Zeigervariablen bestehen zwei Möglichkeiten. Zum einen kann man einen Zeiger auf eine konstante Variable vereinbaren, dann kann zwar der Zeiger verändert werden, nicht jedoch der Inhalt der Variablen, auf die der Zeiger zeigt. Zum zweiten kann ein konstanter Zeiger vereinbart werden, der bei der Definition auf eine Variable gerichtet wird und danach nicht verändert werden darf. Nachfolgendes Beispiel zeigt diese Unterschiede.

Beispiel:

```
int          Variable;
const int    Konst_Var = 10;
const int   *Zeiger_auf_Konst_Var = &Konst_Var;
int * const  Konst_Zeiger_auf_Var = &Variable;
const int * const  Konst_Zeiger_auf_Konst_Var
                                  = &Konst_Var;
```

Symbolische und literale Konstanten

Symbolische Konstanten mit dem **`#define`**-Befehl des Präprozessors vereinbart. Beim Präprozessor-Lauf werden alle Vorkommen symbolischer Konstanten im Programmcode in literale Konstanten aufgelöst. Literale Konstanten haben keinen Namen, sie werden durch ihren Wert dargestellt. Da sie somit nicht geändert werden können, werden sie sinnvollerweise vom Compiler im Code-Segment angelegt.

Beispiel:

```
Zeile 1: #define ANZAHL 100
         ...
Zeile 2: double f[ANZAHL], g[ANZAHL];
         ...
Zeile 3: for (i=0; i<ANZAHL; i=i+1) {
Zeile 4:     f[i]=23.0*g[i];
         }
         ...
```

In Zeile 1 wird eine symbolische Konstante **ANZAHL** vereinbart, die dann in den Zeilen 2 und 3 verwendet (referenziert) wird. Dieses Vorgehen ist sehr hilfreich bei Änderungen, da **ANZAHL** nur an einer Stelle angepasst werden muss.
In Zeile 3 werden die Literalen Konstanten 0 und 1 verwendet, in Zeile 4 die literale Konstante 23.0.

9.2 Attribute

In C existieren die beiden Attribute **const** und **volatile**, die einer Variablendefinition hinzugefügt werden können.

const-Attribut

In obigem Abschnitt 9.1.5 wurde das **const**-Attribut für Variable bereits erläutert.

volatile-Attribut

Das Attribut **volatile** wird häufig mit der Bezeichnung *„Zerbrechlich“* ins Deutsche übersetzt. Mit diesem Attribut werden Speicherzellen gekennzeichnet, welche z.B. zwischen zwei Lesezyklen ihren Wert ändern können.
Beispielsweise hat man dieses Verhalten bei Variablen, auf die zwei asynchrone Softwareprozesse zugreifen (z.B. Semaphore). Oder ein Programm greift auf Register von Peripheriebausteinen zu, die in den Speicherbereich des Prozessors eingeblendet sind (Memory-mapped IO). Aufgrund von Hardwareereignissen kann sich der Inhalt eines solchen Registers ändern.
Moderne optimierende Compiler merken sich jedoch Inhalte von häufig verwendeten Variablen in Registern, um damit Zugriffe auf den externen Speicher zu sparen. Bei **volatile**-Variablen muss der Compiler diese Optimierung abschalten und bei jedem Zugriff auf die Variable im Speicher zugreifen.

Beispiel:

```
volatile int *hardware_reg = 0x03f8
...
while (0==*hardware_reg) { /* HW lesen */
   /* warten */
}
*hardware_reg=0; /* HW schreiben */
```

Durch das **`volatile`**-Attribut muss der Compiler gewährleisten, dass bei jedem Durchlauf der **`while`**-Schleife erneut die Hardware gelesen wird. Ansonsten könnte eine unendliche Schleife entstehen.

9.3 Gültigkeitsbereich von Variablen

Bei der Definition einer Variablen besitzt eine Variable einen Gültigkeitsbereich, er wird im Englischen als *Scope* bezeichnet. In diesem Bereich ist die Variable sichtbar und es kann auf sie zugegriffen werden. Werden mehrere Variable gleichen Namens definiert, die sich in ihren Gültigkeitsbereichen unterscheiden, müssen eindeutige Regeln entscheiden, auf welche Variable zugegriffen wird.

- Innerhalb einer Vereinbarungsebene (z.B. globaler Vereinbarungsteil, Vereinbarungsteil eines Blocks) dürfen Variablennamen nur einmal vergeben werden.
  Beispiel:
  ```
  int main() {
     int x;
     double x; /* Fehler: Gleicher Name ist verboten */
  }
  ```
- Ist auf einer höheren Ebene eine Variable vereinbart, wird diese durch eine Variable gleichen Namens in einer tieferen Ebenen verdeckt. Es wird dann immer auf die Variable in der tieferen Ebene zugegriffen.

Beispiel:

```
#include <stdio.h>
int i=10;
int test();
int main () {
   int i=11;
   { int i=12;
      printf("a: i=%d\n", i);
      printf("b: i=%d\n", test());
   }
   printf("c: i=%d\n", i);
   printf("d: i=%d\n", test());
   return 0;
}
int test(){
   return i;
}
```

Folgende Ausgabe erscheint auf dem Bildschirm:

```
a: i=12
b: i=10
c: i=11
d: i=10
```

Eine Variable ist somit im Block gültig, in dem sie definiert wird. Weiterhin ist sie in hierarchisch darunter liegenden Blöcken gültig, sofern dort keine Variable mit gleichem Namen definiert ist.

9.4 Zusammenfassung der Variablenklassen und ihrer Eigenschaften

Variablentyp	Schlüssel-wort	Gültigkeit	Lebens-dauer	Initiali-sierung	Segment
Automatisch	-	Block	Block	nein	Stack
	auto				
	register				
Lokal, statisch	static	Block	Programm	ja	Daten
Global	static	Datei	Programm	ja	Daten
	-	Programm	Programm	ja	Daten
	extern			-	
Dynamisch	-	(kein Name)	Code-abhängig	nein	Heap
Symbolische und literale Konstanten	-	-	-	-	Code

Grundlagen der Programmierung

Vorlesungsskript der Fachhochschule Osnabrück

Prof. Dr. T. Gervens, Prof. Dr.-Ing. B. Lang, Prof. Dr.-Ing. C. Westerkamp,
Prof. Dr.-Ing. J. Wübbelmann

10 Dynamische Speicherplatzbereitstellung

10.1 Dynamisches Anfordern von Speicherplatz

Statische Felder wie

```
double x[1000];
```

besitzen den Nachteil, dass zur Kompilierzeit die Größe des Feldes festliegen muss. Reicht der die Größe des Feldes für eine Applikation nicht aus, muss die Größe erhöht und das Programm neu kompiliert werden.
Man kann die Anpassung an veränderte Dimensionen durch Verwendung von Präprozessordefinitionen zentral vornehmen:

```
#define DIMFIELD 1000
...
double x[DIMFIELD];
```

Dann kann man überall, wo die Feldgröße benötigt wird, den symbolischen Wert **DIMFIELD** eintragen. Dennoch muss bei einer Änderung von **DIMFIELD** das Programm erneut kompiliert werden.
Wählt man, um spätere Änderungen zu vermeiden, von Anfang an sehr große Felder in einem Programm, führt dies zur Verschwendung von Speicherplatz und trotzdem ist man nie sicher, ob die verwendeten Größen für jedes Feld immer ausreichen.

Abhilfe aus dieser Situation bietet das dynamische Einrichten von Speicherplatz. Es erfolgt zur Programmlaufzeit in zwei Schritten:

1. Vom Betriebssystem wird für das Feld ein zusammenhängender Speicherbereich bestimmter Größe angefordert.
2. Es muss dafür gesorgt werden, dass der eingerichtete Speicherplatz als Feld des gegebenen Typs angesprochen werden kann.

Der erste Schritt erfolgt in C mit der Bibliotheksfunktion **malloc**. Das Interface dieser Funktion ist in *„stdlib.h“* deklariert:

```
void *malloc (unsigned int length);
```

Über die Funktion **malloc** stellt das Betriebssystem Speicherplatz der Größe **length** Bytes zur Verfügung. Die Funktion malloc gibt einen unspezifischen **void**-Zeiger auf den Anfang dieses freien Speicherbereichs der Größe **length** Bytes zurück, der nun vom Programm verwendet werden darf.
Die Größe **length** muss aus der gewünschten Feldgröße und der Größe eines einzelnen Feldelements ab. Sie wird im Programm wie folgt ermittelt:

```
length = <Feldgröße> * sizeof(<Typ>)
```

Kann das Betriebssystem den gewünschten Speicherbereich nicht zur Verfügung stellen, dann liefert **malloc** einen **NULL**-Zeiger, also den Wert 0, zurück.

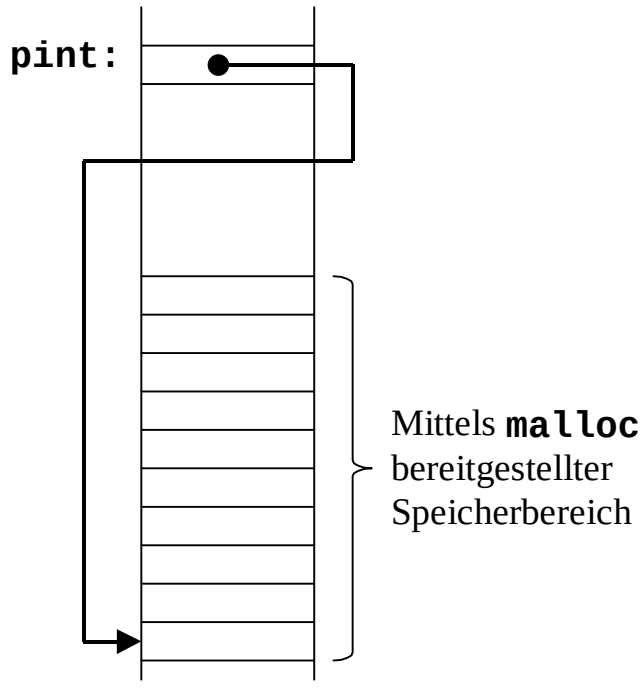
Um den in Schritt 2 bereitgestellten Speicherplatz als Feld eines bestimmten Typs ansprechen zu können, weist man die Anfangsadresse des Bereichs nach einer Typumwandlung mit Hilfe des *cast*-Operators **(**<*Typ*>**)** einer Zeigervariablen vom Typ „<*Typ*>*****“ (Zeiger auf „<*Typ*>“) zu.

10.2 Beispiele: Ein- und zweidimensionale, dynamische Felder

Beispiel: Dynamisches Anfordern eines Speicherbereichs für ein eindimensionales Feld mit *n* Komponenten vom Typ **`int`**:

```
#include <stdlib.h>

int main(){
   int n;
   int *pint;
   ...
   n=1000;
   pint = (int *)malloc(n*sizeof(int));
   ...
}
```

Wird von der **`malloc`**-Funktion für **`pint`** ein gültiger Zeigerwert (kein **`NULL`**-Zeiger) zurückgeliefert, so liegt folgende Situation vor:

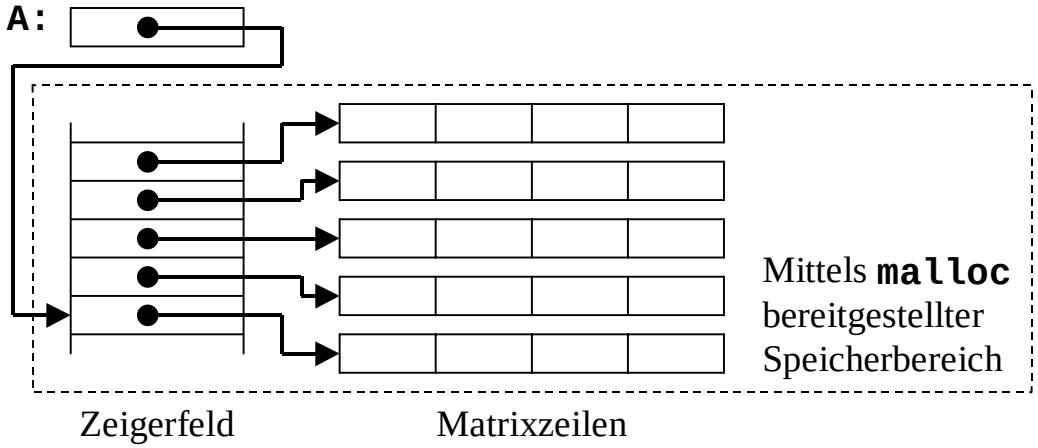


Auf die Komponenten des dynamisch angelegten Feldes kann man genauso zugreifen, wie man es bei statischen Feldern gewohnt ist:

Inhalt der 1-ten Komponente: **`*pint`** oder **`pint[0]`**
Inhalt der 2-ten Komponente: **`*(pint+1)`** oder **`pint[1]`**
usw.

Beispiel: Dynamisches Anfordern eines Speicherbereichs für ein **zweidimensionales** Feld mit *n* Zeilen und *m* Spalten vom Typ **`double`**:

Die dynamische Bereitstellung zweidimensionaler Felder ist komplizierter als der eindimensionale Fall, da *n* Zeilen der Größe *m* (*m* Spalten) dynamisch angelegt werden müssen.
Ein zweidimensionales Feld mit dynamischer Speicherbereitstellung kann mit folgender Struktur aufgebaut werden:



Die zweidimensionale Matrix wird durch einen Zeiger dargestellt, der zunächst auf ein Feld von Zeigern verweist. Jeder Zeiger dieses Zeigerfelds zeigt dann auf eine Matrixzeile. Somit kann über das Zeigerfeld auf die Matrixzeilen zugegriffen werden.

Die Zeigervariable **`A`** muss im Programm als ein „Zeiger auf einen Zeiger" vereinbart werden. Dies wird in C durch zwei Sterne **`**`** gekennzeichnet. Sowohl das Zeigerfeld als auch die Matrixzeilen werden dynamisch mit **`malloc`** angefordert:

```
int i;
double **A;
...
A=(double **)malloc(n*sizeof(double*));
for(i=0; i<n; i++){
   A[i]=(double *) malloc(m*sizeof(double));
}
```

Da die Speicherbereitstellung durch das Betriebssystem vergleichsweise viel Zeit erfordert, erzielt man eine schnellere Laufzeit, wenn man den Speicherplatz für das Zeigerfeld und die Matrixzeilen in einem Block anfordert:

```
A = (double **) malloc (n*sizeof(double*)
                        + n*m*sizeof(double));
A[0]=(double *)(A+n);
for(i=1; i<n; i++) {
   A[i]=A[i-1]+m;
}
```

Auf den Wert der Matrixelemente kann wie im statischen Fall über einen zweidimensionalen Feldzugriff **`A[zeile][spalte]`** oder (bevorzugt) **`*(*(A+zeile)+spalte)`** zugegriffen werden.
Die Zugriffe auf die einzelnen Datenelemente der dynamischen Feldstruktur sind in nachfolgender Tabelle erläutert:

`A`	Zeigervariable, die auf den Anfang des Zeigerfeldes zeigt.
`A+i`	Zeiger auf den Zeiger i im Zeigerfeld.
`*(A+i)` oder **`A[i]`**	Zeiger auf den Beginn der Matrixzeile i.
`*(A+i)+j`	Zeiger auf Element j der Matrixzeile i.
`*(*(A+i)+j)` oder **`A[i][j]`**	Wert von Element j der Matrixzeile i

Beispiel: Programm zum Einrichten eines dynamischen, zweidimensionalen Feldes, dessen Zeilen- und Spaltenanzahl mit **`scanf`** eingelesen werden.

```
#include <stdio.h>
#include <stdlib.h>
int main() {
   int ze, sp, n, m; /* n Zeilen,  m Spalten */
   int **A;
   scanf("%d %d", &n,&m);
   A = (int **)malloc(n*sizeof(int *));
   if (A==NULL) {
       return 1;  /* Fehlerabbruch */
   }
   for(ze=0; ze<n; ++ze) {
      *(A+ze)=(int*)malloc(m*sizeof(int));
      if(*(A+ze)==NULL) {
          return 2;
      }
   }
```

```
   /* Belegen der Komponenten */
   for(ze=0; ze<n; ze++) {
      for(sp=0; sp<m; sp++) {
         *(*(A+ze)+sp)=ze*sp;
      }
   }
   /* Ausdruck der Matrix */
   for(ze=0; ze<n; ze++) {
      for(sp=0; sp<m; sp++) {
        printf("A[%d][%d]=%d", ze, sp, *(*(A+ze)+sp));
      }
      printf("\n");
   }
   getchar();
   return 0;
}
```

10.3 Veränderung der Größe eines bereitgestellten Speicherbereichs

Mit Hilfe der Bibliotheksfunktion **realloc** lässt sich die Größe eines mit **malloc** bereitgestellten Speicherbereichs nachträglich verändern.

```
void *realloc(void *ptr, unsigned int length);
```

Die Funktion **realloc** benötigt zwei Argumente, den Zeiger **ptr** auf einen durch **malloc** bereitgestellten Speicherbereich und die neue Größe **length** des Bereichs in Bytes.

Beispiel:

```
int n;
double *a;
n=50;
a=(double *) malloc(n*sizeof(double));
n=70;
a=(double *) realloc(a, n*sizeof(double));
n=20;
a=(double *) realloc(a, n*sizeof(double));
```

Die Funktion **realloc** sorgt dafür, dass der Inhalt des bisherigen Speicherbereichs erhalten bleibt. Wenn **realloc** direkt hinter dem bisherigen Speicherbereich noch freien Speicherplatz findet, verwendet es diesen und der Zeiger auf den Speicherbereich bleibt erhalten. Ist hinter dem bisherigen Speicherbereich nicht mehr genügend Speicher vorhanden, reserviert **realloc** einen völlig neuen Speicherbereich der neuen Größe und kopiert den Inhalt des bisherigen Bereichs an den Anfang des neuen Bereichs. In diesem Fall ändert sich der Zeiger auf den Speicherbereich und damit auch die Adressen aller Datenobjekte in diesem Bereich.

10.4 Freigabe eines bereitgestellte Speicherbereichs

Wird ein zuvor mit **`malloc`** dynamisch angeforderter Speicherbereich nicht mehr benötigt, kann dieser mit Hilfe der Funktion **`free`** wieder an das Betriebssystem zurückgegeben werden, d.h. frei gegeben werden:

```
void free(void *z);
```

Das Argument von **`free`** muss ein zuvor von **`malloc`** oder **`realloc`** gelieferter Adresswert sein.

11 Strukturen

11.1 Einfache Strukturen

Mit Hilfe von Strukturen können in C Datenobjekte unterschiedlichen Typs zu einer logischen Einheit (Verbund, Record) zusammengefasst werden. Die Datenobjekte bilden dann die Komponenten der Struktur.

Beispiel: Personaldaten

Mit der nachfolgenden Definition werden zwei Strukturvariable mit Namen **person1** und **person2** angelegt, welche mehrere Datenobjekte zur Beschreibung einer Person enthalten:

```
struct {
   char              name[30];
   char              vorname[20];
   short             alter;
   char              abteilung[50];
   unsigned int      eintrittsjahr;
   long int          telefon;
} person1, person2;
```

Allgemeine Syntax:

Werden wenige Variablen einer Struktur benötigt, fasst man die Beschreibung der Struktur und die Definition der zugehörigen Variablen in einer Vereinbarung zusammen:

```
struct {
   <Variablendeklaration1>;
   ...
   <VariablendeklarationN>;
} <Strukturvariable1>,...,<StrukturvariableM>;
```

Werden mehrere Datensätze derselben Struktur benötigt, so deklariert man zunächst die Struktur als einen neuen Datentyp mit der Vergabe eines Strukturtypnamens:

```
struct <Strukturtypname> {
   <Variablendeklaration1>;
   ...
   <VariablendeklarationN>;
};
```

Anschließend definiert man die gewünschten Strukturvariablen unter Verwendung des neuen Datentyps **struct** *<Strukturtypname>*:

```
struct <Strukturtypname> <Strukturvariable>;
```

Eine Struktur fasst somit mehrere Datenobjekte in einer gemeinsamen Datenstruktur zusammen. Diese Datenobjekte werden als **Komponenten** der Struktur bezeichnet.

Beispiel:

```
struct angestellter{
   char          name[30];
   char          vorname[20];
   short         alter;
   char          abteilung[50];
   unsigned int  eintrittsjahr;
   long int      telefon;
   };

struct angestellter person1, person2;
```

Die Strukturvariablen **person1** und **person2** besitzen dann die in **struct angestellter{...}** vereinbarte Struktur.

Initialisierung von Strukturen

Die Initialisierung von Strukturen erfolgt ähnlich wie bei Feldern mit einer nachgestellten Initialisierungsliste:

```
struct {
   char          name[30];
   char          vorname[20];
   short         alter;
   char          abteilung[50];
   unsigned int  eintrittsjahr;
   long int      telefon;
} person1, person2 = { "Mueller", "Heinz", 24,
              "Entwicklung", 2001, 123456 };
```

Zugriff auf die Komponenten von Strukturen

Auf die Komponenten der Struktur kann über den Punkt-Operator, der den Namen der Strukturvariablen mit dem Namen der Komponentenvariablen verbindet, zugegriffen werden:

<Strukturvariable>.<Komponentenvariable>

Dieser Ausdruck kann wie eine normale Variable verwendet werden.

Beispiele:

```
strcpy(person1.name, "Meinert");
person1.telefon=654321;
printf("Eintrittsjahr von %s ein: ", person1.name);
scanf("%d", &person1.eintrittsjahr);
printf("%s in Abteilung %s ist seit %d im
        Unternehmen.\n",
       person1.abteilung,
       person1.name,
       person1.eintrittsjahr);
```

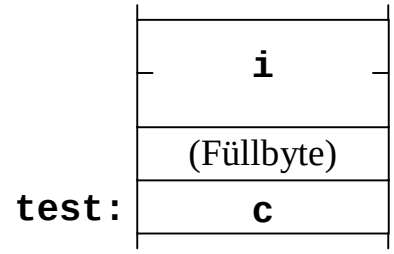
Anordnung von Strukturen im Speicher

Die Komponenten einer Strukturvariablen werden unter Beachtung eines prozessorabhängigen Alignments vom Compiler zusammenhängend im Speicher abgelegt.
Unter Alignment versteht man, dass Datenobjekte abhängig vom Typ nur an bestimmten Adressvielfachen beginnen dürfen. Für *integer*-Datentypen besteht beispielweise meist ein Alignment der Prozessorwortbreite. Bei einer Wortbreite von 16 Bit müssen dann *integer*-Variable immer auf geraden Byte-Adressen beginnen. Folgt in einer Struktur z.B. auf eine *char*-Komponente eine *integer*-Komponente, dann werden hinter der *char*-Komponenten soviele Füllbytes eingefügt, bis das Alignment für die *integer*-Komponente gegeben ist.

Beispiel:

Anordnung einer Struktur **`test`** im Speicher bei 16-Bit Alignment.

```
struct {
   char c;
   int i;
} test;
```



Für die Anfangsadresse einer Struktur gilt meist ebenfalls ein Alignment, welches der Dokumentation des Compilers und des Prozessors zu entnehmen ist.

11.2 Geschachtelte Strukturen

Strukturen können als Komponenten jeden Datentypen, insbesondere auch andere Strukturen enthalten. Dabei können Strukturen beliebig tief geschachtelt werden.

Beispiel:

```
struct betrieb {
   char name[30];
   char ort[30];
};
struct produkt {
   char name[30];
   unsigned int preis;
   struct betrieb B;
};
```

Beispiel:

```
struct punkt{
   double x;
   double y;
};
struct rechteck{
   struct punkt links_oben;
   struct punkt rechts_unten;
};
struct kreis {
   double radius;
   struct punkt mittelpunkt;
};
```

Die Initialisierung geschachtelter Strukturen erfolgt mit geschachtelten Initialisierungslisten.

Beispiel (mit oben vereinbartem Strukturtyp **`struct kreis`**):

```
struct kreis K1={4.0, {1.0,1.0} };
```

Der Zugriff auf die Komponenten geschachtelter Strukturen erfolgt durch mehrfache Anwendung des Punkt-Operators. Dabei müssen die Komponentennamen innerhalb einer Teil-Struktur eindeutig sein.

Beispiel (mit oben definierter Strukturvariabler **`K1`** vom Typ **`struct kreis`**):

Auf die Komponenten von **`K1`** greift man wie folgt zu:

```
K1.radius=8.0;
K1.mittelpunkt.x=10.0;
K1.mittelpunkt.y=10.0;
```

11.3 Operationen mit Strukturen

Mit Strukturvariablen sind 4 Operationen erlaubt:

1. Zuweisung
2. Selektion einer Komponente mit dem Punkt-Operator
3. Ermittlung der Größe mit dem **`sizeof`**-Operator
4. Ermittlung der Adresse mit dem **`&`**-Operator

Weiterhin können sie als Parameter und als Ergebniswert einer Funktion auftreten. Weitere Operationen wie z.B. der Vergleich zweier Strukturen sind nicht erlaubt.

Beispiel: Rechnen mit komplexen Zahlen

Eine Funktion für die komplexe Multiplikation **`mul`** kann wie folgt realisiert werden:

```
struct complex {
   double re, im;
};
struct complex mul (struct complex zahl1,
                    struct complex zahl2) {
   struct complex hilf;
   hilf.re =   zahl1.re*zahl2.re
             - zahl1.im*zahl2.im;
   hilf.im =   zahl1.re*zahl2.im
             + zahl1.im*zahl2.re;
   return hilf;
}
```

Analog lassen sich Funktionen für Addition **`add`**, Division **`div`** und Subtraktion **`sub`** schreiben.

Die komplexe Rechnung *a=x*y-z/y* lässt sich dann wie folgt ausführen:

```
complex a, x, y, z;
a=sub(mul(x, y), div(z, y));
```

Felder von Strukturen

Mehrere Strukturen gleichen Typs können zu Feldern zusammengefasst werden. Jede Komponente des Feldes ist dann eine Struktur.

Beispiel:

Nachfolgende Definition legt Speicherplatz für 100 Strukturen des Typs **`struct complex`** an:

```
struct complex vector[100];
```

Ein Zugriff auf die 54. Feldkomponente erfolgt dann beispielsweise mit:

```
vector[53].re = 0.7;
vector[53].im = 0.2;
```

11.4 Zeiger auf Strukturen

Der Inhalt einer Strukturvariablen vom Typ **X** wird, wie bei allen anderen Objekte, die Speicherplatz belegen, unter einer zugeordneten Adresse im Speicher abgelegt. Diese Adresse kann einer Zeigervariablen vom Typ ‘Zeiger auf Struktur vom Typ **X**’ zugewiesen werden.

Beispiel:

```
struct stud {
   char  name[30];
   char  vorname[20];
   short semester;
};
struct stud Person1, *pPerson;
pPerson=&Person1;
```

Um über den Zeiger auf die Komponenten zuzugreifen, kann der Inhaltsoperator verwendet werden:

```
printf("%s studiert im %d Semester.",
       (*Person).name,(*pPerson).semester);
```

Der -> Operator

Da die Klammerschreibweise umständlich ist, gibt es in C den **->** Operator als Abkürzung. Beispielsweise kann statt:

```
(*pPerson).name
```

alternativ auch

```
pPerson->name
```

geschrieben werden. Allgemein kann also mit:

<Strukturzeiger>->*<Komponentenname>*

über einen Zeiger auf die Komponente einer Struktur zugegriffen werden.

Einbinden von Komponenten des eigenen Typs

Strukturen dürfen einen Zeiger auf den eigenen Strukturtyp enthalten, nicht aber eine Komponente vom eigenen Typ.

Beispiel:

```
struct richtig {
   struct richtig *pA; /* erlaubt */
};
struct fehlerhaft {
   struct fehlerhaft A; /* nicht erlaubt !!! */
};
```

Somit ist die Verwendung des eigenen Strukturtyps zur Vereinbarung eines Zeigers schon zu einem Zeitpunkt möglich, an dem die Deklaration des Typs selber noch nicht abgeschlossen ist.

Gegenseitiger Verweis zweier Strukturen aufeinander

Benötigt man zwei Strukturtypen, die gegenseitig über Zeiger aufeinander verweisen, so muss man einen Strukturtyp schon bekannt machen, bevor er deklariert ist. Dies erfolgt mit einer **unvollständigen Typdeklaration**.

Beispiel:

Es sollen die Datenstrukturen flip und flop vereinbart werden, die gegenseitig aufeinander verweisen:

```
struct flip;  /* Unvollständige Typdeklaration */
struct flop { /* Deklaration von struct flop */
   struct flip *f;
};
struct flip { /* Deklaration von struct flip */
   struct flop *f;
};
```

Im selben Gültigkeitsbereich, in dem die unvollständige Typdeklaration erfolgt, muss auch die vollständige Deklaration durchgeführt werden.

Beispiel: Vollständiges Programm

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

int main () {
   int  i ;
   struct person {
      char  name [20];
      char  vorname [20];
   };
   struct student {
      struct  person  name;
      float   zensur;
   };
   struct student stud , *stud_p;
   struct student gruppe_WS02[40];

   strcpy (stud.name.name,    "Meiser");
   strcpy (stud.name.vorname, "Hans");
   stud.zensur = 1.3f;
```

```
    stud_p = (struct student *)
                  malloc(sizeof(struct student));
    if (stud_p == NULL) {
       puts ("Fehler beim Allokieren");
       return 1;
    }
    strcpy ((*stud_p).name.name,  "Muster");
    strcpy (stud_p->name.vorname, "Wilhelm");
    stud_p->zensur = 2.0f;

    gruppe_WS02[0] = stud;
    gruppe_WS02[1] = *stud_p;
    for (i = 0 ; i < 2 ; i++) {
       printf ("Zensur fuer %s %s:  %3.1f\n",
               gruppe_WS02[i].name.vorname,
               gruppe_WS02[i].name.name,
               gruppe_WS02[i].zensur);
    }
    free (stud_p);
    return 0;
}
```

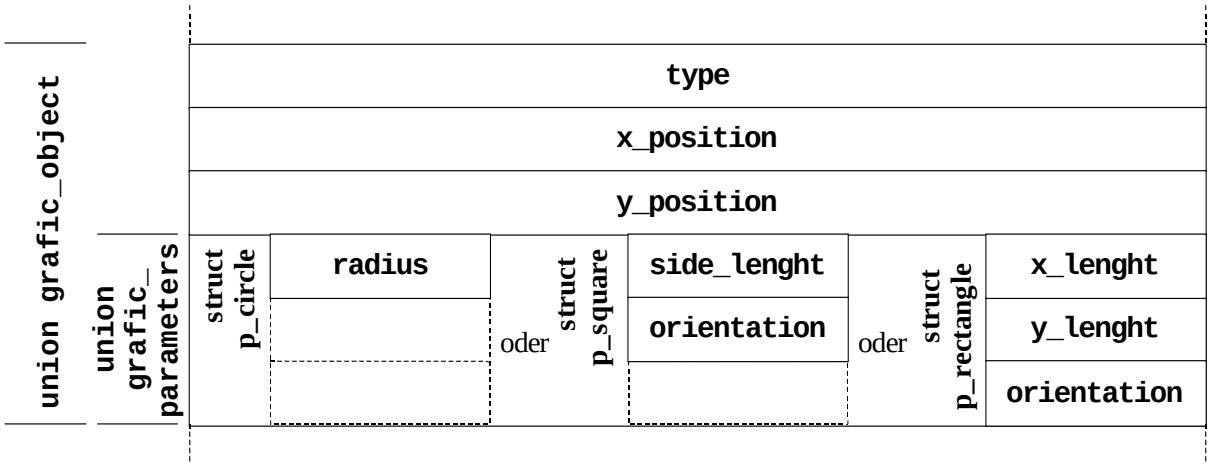
11.5 **union**-Datenstruktur

Ergänzend zur **struct**-Datenstruktur können in C über eine **union**-Datenstruktur eigene zusammengesetzte Typen vereinbart werden. Die Form entspricht der **struct**-Vereinbarung, es wird jedoch das Schlüsselwort **union** verwendet. Im Unterschied zum **struct** werden bei einer **union**-Vereinbarung alle spezifizierten Elemente im Speicher auf den gleichen Speicherplatz gelegt. Sie stellen somit eine Alternative dar. Bei Verwendung einer **union**-Datenstruktur sollte somit immer nur eines der Elemente angesprochen werden.

Beispiel: Vereinbarung einer Datenstruktur für Grafikobjekte

```
struct grafic_object {
  int type; /* 0-Point,  1-Circle,
               2-Square, 3-Rectangle */
  int x_position;
  int y_position;
  union grafic_parameters {
    struct p_circle {
      int radius;
    } circle;
    struct p_square {
      int side_length;
      int orientation;
    } square;
    struct p_rectangle {
      int x_length;
      int y_length;
      int orientation;
    } rectangle;
  } param;
};
```

Anordnung im Speicher:



Zugriff:

```
struct grafic_object g1;
g1.type = 2; /* square */
g1.x_position=10; g1.y_position = 20;
g1.param.square.side_length = 100;
g1.param.square.orientation = 45;
```

Achtung: Der Zugriff auf andere Elemente der **union**-Struktur wird nicht überwacht. Folgende Zugriffe sind möglich, aber vermutlich nicht gewünscht:

```
struct grafic_object g2;
g2.type = 1; /* circle */
g2.x_position=15; g1.y_position = 29;
g2.param.rectangle.x_length = 100;    /* !! */
g2.param.rectangle.y_length = 15;     /* !! */
g2.param.rectangle.orientation = 180; /* !! */
```

Damit wird im Beispiel der Radius indirekt auf 100 gesetzt.

Die Überwachung der Konsistenz der **union**-Datenstruktur obliegt alleine dem Programmierer. Von Seiten C wird keine Konsistenzüberprüfung vorgenommen.

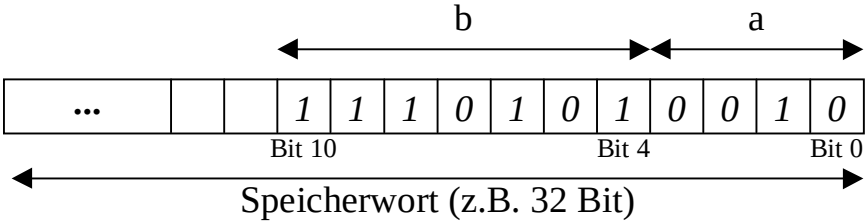
11.6 Bitfelder

Eine besondere Komponente der **struct-** und **union-**Datenstrukturen ist das Bitfeld. Es erlaubt den Zugriff auf einzelne Bits eines Speicherwortes.
Als Bitfeld-Typ ist üblicherweise nur der **int**-Datenyp als **signed** oder **unsigned** erlaubt, manche Compiler erlauben auch die Verwendung vom Typ **char**.
Bitfelder sind zur Unterstützung von Hardware gedacht, wo in Peripheriebausteinen einzelne Bitgruppen gelesen und geschrieben werden müssen.

Beispiel: Nachfolgender Code zeigt die Verwendung von Bitfeldern:

```
struct Bitfeld_Beispiel {
  unsigned int a:4;
  signed   int b:7;
};

struct Bitfeld_Beispiel bf;

bf.a=2;   /* 0010 */
bf.b=-11; /* 1110101 */
```

Im Speicher werden 4+7=11 Bits in einem Speicherwort belegt:



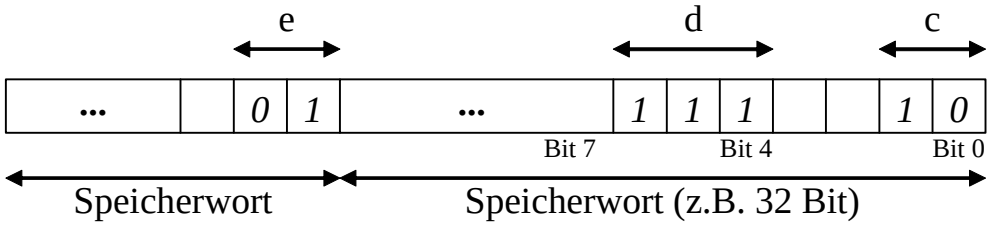
Die Komponente a wird als positive Zahl mit 4 Bit, die Komponente b als ganze 7-Bit Zahl in 2er-Komplementdarstellung verwendet.

Werden Lücken zwischen zwei Bitfeldern benötigt, können Komponenten ohne Namen eingefügt werden. Muss ein Bitfeld am Wortanfang stehen, wird vorher ein Bitfeld der Länge 0 eingefügt.

Beispiel: Lücken zwischen Bitfeldern, Wort-Alignment.

```
struct Bitfeld_Beispiel {
  unsigned int c:2;
  unsigned int  :2;
  signed   int d:3;
  unsigned int  :0;
  unsigned int e:2;
};

struct Bitfeld_Beispiel bf;

bf.c =  2;  /* 10  */
bf.d = -1;  /* 111 */
bf.e =  1;  /* 01  */
```

Im Speicher werden Bits in zwei Speicherworten belegt:



Es ist zu beachten, dass die Implementierung von Bitfelder sehr stark von dem jeweils verwendeten Rechner abhängen. Daher sollte man beim Erstellen portabler Programme, die auf unterschiedlichen Rechnern und Prozessoren ablaufen sollen, auf die Verwendung von Bitfeldern verzichten. Die Implementierung von Bitoperationen kann dann alternativ mit den bitweisen Logikoperatoren (**&**, **|**, **~**, **^**) erfolgen.

12 Aufzählungstypen

Mit dem **enum**-Schlüsselwort (engl.: enumeration) lassen sich Aufzählungstypen vereinbaren:

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

Mit dem Beispiel wird ein neuer Datentyp **enum Boolean** vereinbart, dem die Aufzählungskonstanten **FALSE** und **TRUE** zugeordnet werden.

Die Aufzählungskonstanten haben einen ganzzahligen Wert vom Typ **int**. Werden den Aufzählungskonstanten keine Werte zugeordnet, erhalten sie mit 0 beginnend aufeinanderfolgende Integerwerte. In obigem Beispiel ist somit der Wert von **FALSE** 0 und von **TRUE** 1.

Den Aufzählungskonstanten können auch explizit Werte zugeordnet werden:

```
enum Zahlen = {EINS=1, ZWEI, VIER=4, DREI=3};
```

Werden Konstanten nicht explizit belegt (im Beispiel die Konstante **ZWEI**), erhalten sie den um 1 erhöhten Wert der vorhergehenden Konstante zugeordnet.

Im selben Gültigkeitsbereich darf eine Aufzählungskonstante nur in einer **enum**-Typvereinbarung vorkommen. Auch darf der Name nicht mit einem Variablennamen im gleichen Bereich übereinstimmen.

Die Vereinbarung einer **enum**-Variablen erfolgt durch Angabe des **enum**-Typs gefolgt von dem Variablennamen:

```
enum Boolean Schalter;
enum Zahlen Test;
```

Das Ansprechen von **enum**-Variablen erfolgt wie mit integer-Variablen. Man kann einer **enum**-Variablen auch integer-Werte zuweisen. Eine Überprüfung auf den spezifizierten Wertebereich erfolgt nicht.

```
Schalter = TRUE;
Test     = VIER;
Schalter = 27;   /* Erlaubt aber unschoen */
Test     = 11;   /* Erlaubt aber unschoen */
```

13 Typdefinitionen

In C können mit Hilfe der **`typedef`**-Anweisung neue Typnamen definiert werden. Dabei werden jedoch keine neuen Variablentypen geschaffen, sondern vorhandene Typen mit einem neuen Namen versehen.
Solch einen neuen Namen bezeichnet man auch als Alias-Namen (alias: eng.: Deckname).
`typedef` verhält sich ähnlich wie **`#define`**, nur dass der Textersatz vom Compiler übernommen wird. Dabei überprüft der Compiler auch sofort, ob der verwendete Typname korrekt angewendet wird.
Durch den Ersatz einer komplizierten Typvereinbarung in C durch einen neuen Namen lassen sich komplizierte Datentypen in C übersichtlicher darstellen.

Beispiel:

```
typedef int TYP_INT;
TYP_INT i, j, k, l;
```

Der Typname **`TYP_INT`** kann wie der Typ **`int`** genutzt werden.

Beispiele:

```
typedef int *PINT; /* PINT ist Synonym für int* */
PINT k;            /* k ist Zeiger auf int. */

typedef struct {
   char ort[40];
   int plz;
   char *land;
   long int anzahl;
} LAND_STRUKTUR;
LAND_STRUKTUR Niedersachsen, Nordrheinwestfalen;

typedef int *(*FP)(char *,char *);
FP f; /* f ist Zeiger auf eine Funktion mit
         Rückgabewert Zeiger auf int und
         2 Parametern vom Typ char* */
FP (*g)[10]; /* Zeiger auf ein Feld mit
                10 Komponenten vom Typ FP */
/* Gleichartige Definition ohne typedef */
int *(*(*h)[10])(char *,char *);
```

Verabredungsgemäß werden Typnamen **GROSS** geschrieben. Dies ist jedoch keine Vorgabe des Compilers sondern nur eine Vereinbarung. Diese Vereinbarung bietet den Vorteil, dass man sofort selbstdefinierte Typen in einem Programmcode erkennt.

14 Direkte Sprungbefehle

Im Kontrast zum strukturierten Ansatz von C steht die Sprunganweisung **goto**. Sie erlaubt beliebige Sprünge innerhalb einer Funktion. Ein Sprung aus einer Funktion in eine andere Funktion ist mit der **goto**-Anweisung nicht möglich.
Die Sprunganweisung benötigt eine Marke (Label) als Sprungziel. Eine Marke kann vor jede ausführbare Anweisung gesetzt werden. Sie wird durch einen Namen gefolgt von einem Doppelpunkt deklariert.

Beispiel: Überspringen einer Anweisung

```
#include <stdio.h>
int main(){
  goto MacheNichts;      /* Sprung zur Marke*/
  printf("Hello World\n");
  MacheNichts: return 0; /* Marke vereinbaren */
}
```

Bei obigem Code wird die **printf**-Anweisung nicht ausgeführt.

Da die **goto**-Anweisung dem Gedanken der strukturierten Programmierung widerspricht, sollte sie bei der Implementierung von Algorithmen keine Verwendung finden. Hilfreich ist diese Anweisung jedoch, wo die algorithmische Struktur aufgrund von Fehlern verlassen werden muss und dies mit break und continue nur umständlich möglich ist. In diesem Fall kann direkt von der fehlerhaften Stelle des Algorithmus in eine Fehlerbehandlung gesprungen werden.

Beispiel: Fehlerbehandlung ohne und mit **goto**-Anweisungen

Nachfolgendes Code-Beispiel zeigt eine Überprüfung der Fehler im Algorithmus und eine konzentrierte Fehlerbehandlung am Ende der Funktion. Man erkennt, dass durch die Fehlerbehandlung die Schachtelungstiefe im Algorithmus mit jeder Fehlerüberprüfung zunimmt.

Man könnte auch die Fehlerbearbeitung verteilt im Algorithmus durchführen und die Funktion im Fehlerfall im Inneren des Algorithmus mit **return** verlassen.

Beide Lösungen sind nicht sonderlich zufriedenstellend.

Das zweite Codebeispiel zeigt die Verwendung der **goto**-Anweisung im Fehlerfall. Sobald ein Fehler entdeckt wird, wird die Ursache des Fehlers in einer Fehlervariablen gespeichert und ein völlig separater Fehlerbehandlungsteil der Funktion mittels **goto Fehler** angesprungen. Damit ist eine saubere Trennung zwischen dem zu realisierenden Algorithmus und der Fehlerbehandlung möglich.

Codebeispiel 1: Fehlerbehandlung innerhalb des Algorithmus

```
#include <stdio.h>

typedef enum fehler_e {
  KEIN_FEHLER = 0,
  ZUWENIG_ARGUMENTE,
  KANN_IN_NICHT_OEFFNEN,
  KANN_OUT_NICHT_OEFFNEN
} FEHLER;

int main(int argc, char*argv[]) {
  FILE   *in, *out;
  FEHLER error = KEIN_FEHLER;
  if (argc<3) {
    error = ZUWENIG_ARGUMENTE;
  } else {
    in=fopen(argv[1],"rb");
    if (NULL==in) {
      error = KANN_IN_NICHT_OEFFNEN;
    } else {
      out=fopen(argv[2],"wb");
      if (NULL==in) {
        error = KANN_OUT_NICHT_OEFFNEN;
      } else {
        ... /* Programm weiter ausfuehren,
               aber jede Fehlerbehandlung
               erhoeht die Schachtelung */
      }  /* of 3rd else */
    }   /* of 2nd else */
  }   /* of 1st else */
  if (error) {
    /* Fehlerbehandlung */
    switch(error) {
      case ZUWENIG_ARGUMENTE:
        ...
        break;
      case KANN_IN_NICHT_OEFFNEN:
        ...
        break;
      case KANN_OUT_NICHT_OEFFNEN:
        ...
        break;
    }
  }
  return (int) error;
}
```

Codebeispiel 2: Fehlerbehandlung mit goto

```
#include <stdio.h>

typedef enum fehler_e {
  KEIN_FEHLER = 0,
  ZUWENIG_ARGUMENTE,
  KANN_IN_NICHT_OEFFNEN,
  KANN_OUT_NICHT_OEFFNEN
} FEHLER;
int main(int argc, char*argv[]) {
  FILE   *in, *out;
  FEHLER error = KEIN_FEHLER;
  if (argc<3) {
    error = ZUWENIG_ARGUMENTE;
    goto Fehler;
  }
  in=fopen(argv[1],"rb");
  if (NULL==in) {
    error = KANN_IN_NICHT_OEFFNEN;
    goto Fehler;
  }
  out=fopen(argv[2],"wb");
  if (NULL==in) {
    error = KANN_OUT_NICHT_OEFFNEN;
    goto Fehler;
  }    /* ... Programm weiter ausfuehren,
                Fehlerbehandlung erhoeht
                nicht mehr die Schachtelung */
  return 0;
 /* ------ Fehlerbehandlung ------ */
Fehler:
  switch(error) {
    case ZUWENIG_ARGUMENTE:
      /*...*/
      break;
    case KANN_IN_NICHT_OEFFNEN:
      /*...*/
      break;
    case KANN_OUT_NICHT_OEFFNEN:
      /*...*/
      break;
  }
  return (int)error;
}
```

Grundlagen der Programmierung

Vorlesungsskript der Fachhochschule Osnabrück

Prof. Dr. T. Gervens, Prof. Dr.-Ing. B. Lang, Prof. Dr.-Ing. C. Westerkamp,
Prof. Dr.-Ing. J. Wübbelmann

15 Datenstrukturen

Analog zu Kontrollstrukturen sind für die Entwicklung eines Programms die **Datenstrukturen** wichtig. Datenstrukturen beschreiben nicht nur die **Inhalte,** sondern auch die **Beziehungen** (Relationen) der Daten untereinander.

Beispiel: Ein Stammbaum besteht aus Namen inkl. Beziehungen

Wichtige Datenstrukturen sind Felder, Listen, Warteschlangen, Stapel- und Baumstrukturen.

15.1 Felder

Ein (lineares) Feld ist eine Reihung einer festen, geordneten Menge von Datenelementen. Jedes Element (mit Ausnahme des letzten) hat einen Nachfolger und (bis auf das erste Element) einen Vorgänger. Weitere Namen solcher Felder sind Arrays und Vektoren.

```
#define MAX_LEN 100

struct element {
    int info_int;
    char info[40];
} Feld[MAX_LEN];
```

Jedes Element wird durch seine Position bzw. seinen Index identifiziert und angesprochen. Häufig geschieht der Zugriff über Zeiger.
Da alle Elemente im Hauptspeicher unmittelbar aufeinander folgend abgelegt sind, spricht man auch von **sequentieller** Speicherung.

15.2 Listen

Eine Liste ist ein Reihung von beliebig vielen Datenelementen mit Vorgänger und Nachfolger wie bei einem Feld. Die Datenelemente stehen jedoch nicht aufeinander folgend im Hauptspeicher, sondern werden dynamisch angelegt. Jedes Element enthält einen Zeiger auf seinen Nachfolger (**einfach verkettete Liste**). Man spricht daher auch von rekursiven Datenstrukturen.
Da mit der eigentlichen Information immer auch ein Zeiger auf das nächste Element gespeichert werden muss, verwendet man Strukturen für die Datenelemente.

```
struct element {
   int info;                        /* info */
   struct element *naechster;
} *start;
```

Häufig stehen die Operationen

`push()` für das Einfügen und

`pop()` für das Entfernen von Elementen am Ende der Liste zur Verfügung.

Enthält die Liste zusätzlich einen Verweis auf den Vorgänger, dann spricht man von **doppelt verketteten Listen**.

```
struct element {
   int info;
   struct element *naechster;
   struct element *vorgaenger;
} *start;
```

Der Vorgänger des ersten und der Nachfolger des letzten Elements sind in der Regel NULL-Zeiger.
Doppelt verkettete Listen sind besonders geeignet für Durchlaufen einer Liste in Vorwärts- und Rückwärts-Richtung.
Mit auf **`push()`** und **`pop()`** aufbauenden Operationen kann z.B. eine Liste sortiert werden.

Beispiel 1:

```
/*  Definition einer "rekursiven Struktur", eine einfache
 "verkettete Liste" mit Dateizugriff
 (Programm struct4.c)
 ================================================
 Demonstriert werden

 *  die Definition einer "rekursiven Struktur",
 *  das "Verketten" von Strukturen mit Pointern,
 *  Mehrfachzuweisungen.
 */

#include <string.h>
#include <stdio.h>
#include <stdlib.h> /* ... fuer 'malloc' und 'free' */
typedef  struct p_tag  {
                char  name    [50] ;
                char  vorname [50] ;
                float zensur       ;
                struct p_tag *next ; }student ;

student *push (char *, char *, float);
student *datei_lesen(char *, student *);
void ausgabe(student *);

/* Hauptprogramm */

int main (int argc, char **argv){
    student   *root_p, *stud_p;

    char s1[50], s2[50];
    float note;
    int i;

    root_p = push("Korn","Klara",1.7f);
    stud_p=root_p;
/* Mit stud_p=stud_p->next=push wird die Adresse des
 * neuen Elementes am Ende der Liste eingetragen und der
 * aktuelle Zeiger stud_p auf dieses Element gesetzt.
 */
    stud_p=stud_p->next=push("Cron","Maria",2.7);
    stud_p=stud_p->next=push("Walker","John",1.3f);
    stud_p=stud_p->next=push("Beam","Jim",2.0f);
```

```
/* Einlesen aus einer Datei */
  if(!argv[1]) {
      printf("Adressen-Datei uebergeben!");
      exit(0);
  }
  else {
    stud_p=datei_lesen(argv[1], stud_p);
  }

/* Einlesen ueber Tastatur */
printf("Moechten Sie weitere Daten eingeben?(j/n)\n");
  fflush(stdin);
  while((i=getchar())=='j') {
      scanf("%s%s%f",s1,s2,&note);
      stud_p = stud_p->next = push (s1,s2, note) ;
  printf("Weiter(j/n)");
  fflush(stdin);
}

/* die "verkettete Liste" wird ausgegeben   */

  ausgabe(root_p);

/* Freigabe des allokierten Speicherplatzes   */

  stud_p = root_p;
  while (stud_p != NULL) {
       root_p = stud_p->next ;
       free (stud_p) ;
       stud_p = root_p ;
    }
    return 0;
}

/* Unterfunktion zum Allokieren von Speicherplatz und
   Einfügen eines Elementes */
student *push(char *nachname, char *vorname, float zens) {
   student *stud_p;

   if((stud_p=(student *)malloc(sizeof (student)))==NULL){
       puts ("Fehler beim Allokieren von Speicherplatz");
       exit (1);    /*  Ende, kein Speicher */
    }
```

```
    strcpy (stud_p->name    , nachname)  ;
    strcpy (stud_p->vorname , vorname) ;
    stud_p->zensur = zens ;
    stud_p->next   = NULL ;
    return  stud_p;
}

/* Unterfunktion um datei einzulesen */

student *datei_lesen(char *datei, student *stud_p) {
    char s1[50], s2[50];
    float note;
    FILE *fp;
    int i, nl, c;  /* nl: Anzahl der Zeilen */

    fp=fopen(datei,"r");
        if(!fp)printf("\nKeine Datei\n");
        nl=0;
        while((c=getc(fp))!=EOF) {
            if(c=='\n')   nl++;
        }
        rewind(fp);
        for(i=1; i<=nl; i++) {
            fscanf(fp, "%s%s%f", s1, s2, &note);
            stud_p = stud_p->next = push (s1,s2, note);
        }
    fclose(fp);
    return stud_p;
}

/* Unterfunktion zur Ausgabe auf dem Bildschirm */
void ausgabe(student *wurzel) {
    student *sp;
    sp = wurzel;

    while (sp != NULL) {
        printf("%-20s%-5s:\t\t%f\n",
                sp->name, sp->vorname, sp->zensur);
        sp = sp->next ;
    }
}
```

Beispiel 2: Verkettete Liste

```
/*  eine einfach verkettete Liste                  */
/*  Elemente werden beim Hinzufügen sortiert        */

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

struct Element {
    char  s[101];       /* Infodaten: ein Text   */
    struct Element  *n; /* Verweis auf Nachfolger */
};

struct Element *einfuegen(char s[], struct Element *a);
struct Element *loeschen (char s[], struct Element *a);
void  anzeigen(struct Element *a);

void main(){
    struct Element * dieListe=NULL; /* der Listenkopf */
    char    s[101]; /* ein Text      */
    int befehl;

    printf("\n    Einfuegen(1), Loeschen(2),
              Anzeigen(3),  Ende(9) >");

    while(EOF!=scanf("%d",&befehl)){
        getchar();       /* '\n' war noch im Puffer */

        switch(befehl) {
        case 1 : printf("Text zum Einfuegen >"); gets(s);
                 dieListe=einfuegen(s,dieListe);
                 break;
        case 2 : printf("Text zum Loeschen >"); gets(s);
                 dieListe=loeschen(s, dieListe);
                 break;
        case 3 : anzeigen(dieListe);
                 break;
        case 9 : printf("ENDE\n");
                 return;  /* Ende */
        } /* of switch */
```

```
    printf("\nEinfuegen(1), Loeschen(2),
              Anzeigen(3), Ende(9) >");
} /* end of while */

} /* End of main */


struct Element *einfuegen(char s[], struct Element *a){

  struct Element *p=a;  /* aktuelle Position   */
  struct Element *vp;   /* und Vorgaengerposition  */
  struct Element *neu;  /* Zeiger auf neues Element */

/* Einrichten eines neuen Elementes */
                      /*1. Speicherplatz anfordern */
  neu=malloc(sizeof(struct Element));
  strcpy((*neu).s,s);/*2. mit Nutzdaten fuellen   */
                      /*neues Element an den Anfang ?*/
  if(p==NULL || strcmp(s,(*p).s)<0){
    a=neu;          /* neues Element wird Listenkopf */
    (*neu).n=p;     /* bisheriges erstes Element   */
    return a;       /* wird dessen Nachfolger   */
  }
                   /* sortiergerechte Einfuege- */
                   /* position suchen           */
  while(p!=NULL && strcmp(s,(*p).s) >= 0) {
    vp=p;                 /* alte Position merken */
    p=(*p).n;             /* zum naechsten Element*/
  }

  (*neu).n=p;         /* Einhaengen in Verkettung */
  (*vp).n=neu;
  return a;
}



void  anzeigen(struct Element *a){
  while(a!=NULL){
    printf("\n%s",a->s);
    a=a->n;
  }
}
```

```
struct Element *loeschen (char s[], struct Element *a){
  struct Element *p=a;  /* aktuelle Position  */
  struct Element *vp;   /* und Vorgaengerposition   */

                        /* 1. Element loeschen ?    */
  if(p!=NULL && strcmp(s,(*p).s)==0){
    a=a->n;      /* 2. Element wird nun 1. Element */
    free(p);     /* Speicherplatz des alten       */
                 /* Elementes freigeben           */
    return a;
  }

                    /* zu loeschendes Element suchen */
  while(p!=NULL && strcmp(s,(*p).s) > 0) {
    vp=p;                /* alte Position merken */
    p=p->n;              /* zum naechsten Element*/
  }

  if(p!=NULL && strcmp(s,(*p).s)==0) {/* Treffer? */
    vp->n=p->n;             /* neue Verkettung */
    free(p);        /* Speicherplatz freigeben */
  }
  return a;
}
```

15.3 Stapel (Stack)

Ein Stapel (Stack) ist ein Feld oder eine Liste, in der jedoch die Operationen Einfügen und Löschen nur am oberen Ende (top) vorgenommen werden (**LIFO**-Liste).

Beispiele:
- Abstellgleis der Bahn
- Systemstapel zur Abarbeitung von Funktionen
- Taschenrechner mit UPN

Ein Stapel kann entweder als sequentielles Feld mit Hilfe eines Feldes oder dynamisch hinsichtlich der Speicherplatzbelegung als einfach oder doppelt verkettete Liste programmiert werden.

15.4 Warteschlange (queue)

Die Warteschlange ist eine geordnete Liste, in der alle Einfügungen am unteren Ende und alle Löschungen am oberen Ende stattfinden (**FIFO-Liste**).

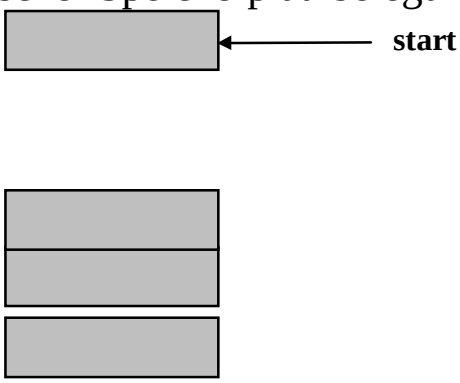
Beispiel:
- Job-Abarbeitung (z.B. Druckaufträge) durch das Betriebssystem,
- Warteschlange an der Kaufhauskasse

Eine Warteschlange kann sequentiell als Feld oder dynamisch hinsichtlich der Speicherplatzbelegung als (einfach oder doppelt) verkettete Liste programmiert werden.

```
/* Stack-Funktionen */

#define MAX_STACK 100
static struct element  {
        int key;
};
struct element stack[MAX_STACK];
int top=-1;

void push(int item)
{
if(top >=MAX_STACK-1){
   printf("Stapelueberlauf! Kein Platz mehr.");
   return;
}
else {
    stack[++top].key=item;
    return;
   }
}

int delete(void) {
if(top==-1) {
    printf("\nStack ist leer.\n");
    return -1;
}
return stack[top--].key;
}

void stapel_aus(void) {
int i;
if(top==-1)
       {    printf("\nStack ist leer.\n");
        return;
       }
for(i=top;i>=0;i--)
    printf("%d: %d\n",i,stack[i].key);
return ;
}

/* Funktionen einer Warteschlange */
#define MAX_STACK 100
static struct element {
```

```
int key;
};
struct element queue[MAX_STACK];
int top=-1, bottom=-1;

void push(int item) {
if(top >=MAX_STACK-1)
 {printf("Stapelueberlauf! Kein Platz mehr.");
  return;
 }
else   {
    queue[++top].key=item;
    return;
   }
}
int delete(void) {
if(top==bottom)   {
    printf("\nQueue ist leer.\n");
    return -1;
    }
else
   return queue[++bottom].key;
}

void stapel_aus(void){
int i;
if(top==bottom)
       {    printf("\nSchlange ist leer.\n");
        return;
    }
for(i=bottom+1;i<=top;i++)
    printf("%d: %d\n",i,queue[i].key);
return ;
}
```

Beispiel: Aufbau eines Stacks (LIFO) als einfach verkettete Liste mit dynamischer Speicherplatzbelegung




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

static struct knoten{
int element;
struct knoten *naechster;
} *start,*p;    /* p Hilfs-Zeiger ; start zeigt
                   auf den Anfang des     Stacks.*/

int push(int wert){
p=(struct knoten *)malloc(sizeof(struct knoten));
if(p==NULL)   {
    printf("Stapelueberlauf! Kein Speicher mehr.");
    return 1;
}
else   {
    p->naechster=start;
    p->element=wert;
    start=p;
   }
return 0;
}

int pop(void) {
    if(start == NULL) {
        puts("Stapel ist leer");
        return 0;
    }
    else {
        printf("Element mit Wert %d
                entnommen.\n",(*start).element);
        p=start;
        start=start->naechster;
```

```
        free(p);
    }
    return 1;
}

int stapel_aus(void){
    struct knoten *start_aus_ptr=start;
    if (start_aus_ptr==NULL) {
        puts("Stapel ist leer\n");
        return 1;
    }
    else {
      while(start_aus_ptr!=NULL) {
        printf("%d\t",(*start_aus_ptr).element);
        start_aus_ptr=start_aus_ptr->naechster;
    }
    printf("\n");
    return 0;
  } /* of int stapel_aus */
}

int main(void) {
    int zahl, c;

    do {
       printf("Stapelfunktionen: push (1),
             pop (2), ausgabe (3), exit (4)\n");
        scanf("%d",&c);0
        if(c==1){
            puts("\n Bitte Zahl eingeben");
            scanf("%d",&zahl);
            push(zahl);
            stapel_aus();
        }
        if(c==2){
            pop();
            stapel_aus();
        }
        if (c==3) {
            stapel_aus();
        }

    } while(c==1||c==2||c==3);
```

```
    if(c==4) {
        return 0;
    }
    return 0;
}
```

15.5 Bäume

Bäume stellen eine wichtige **nichtlineare** Datenstruktur dar. Sie treten häufig in Programmen auf, in denen hierarchische Strukturen eine Rolle spielen.

Die Datenelemente werden im Baum als **Knoten**, die Relationen im Baum als **Äste (Zweige, Kanten)** dargestellt. Den Knoten auf der höchsten Ebene nennt man Wurzel.

Wichtige Eigenschaften von Bäumen:

- Bis auf die Wurzel haben alle Knoten einen **Vorgänger**.
- Bis auf die Endknoten (Blätter) haben alle Knoten **Nachfolger**.
- Zwei Knoten in einem Baum können genau durch einen Pfad verbunden werden.
- Ein Baum mit n Knoten hat n-1 Kanten.
- Jeder einzelne Knoten eines Baums kann selbst wieder als Wurzel eines Teilbaums angesehen werden (**rekursive Datenstruktur**).

Beispiele:

- Organigramm eines Unternehmens
- Darstellung einer Anlage aus Anlageteilen
- Funktionsstrukturierung in einem Programm
- Stammbaum

Binärbaum

Eine Sonderform stellt der Binärbaum dar. Bei ihm gehen von jedem Knoten höchstens zwei Zweige aus. Durch ihren einfachen Aufbau eignen sie sich gut zum Sortieren.
Ein Knoten eines Binärbaumes besteht aus einem Informationsteil (Daten oder Zeiger auf Daten) und je einem Zeiger auf den linken und rechten Nachkommen. Wie bei der doppelt verketteten Liste ist es möglich, mit einem weiteren Zeiger auf den Vorfahr zu verweisen. Ein fehlender Nachkomme ist mit einem NULL-Zeiger gekennzeichnet.

```
struct knoten {
    char wort[200];
    struct knoten *left;
    struct knoten *right;
};

struct knoten *root;
```

Alle Methoden/Operationen wie sortieren, durchlaufen, suchen etc. mit Bäumen nutzen ihre **Rekursivität**.

Es gibt verschiedene Baumordnungen, die dabei gebräuchlich sind:

- **preorder** (zuerste die Wurzel, dann die Unterbäume von links nach rechts)
- **postorder** (zuerst die Unterbäume von links nach rechts, dann die Wurzel)
- **inorder** (zuerst der Unterbaum des linken Nachkommen, dann die Wurzel, dann die anderen Unterbäume von links nach rechts)

```
void bearb_pre(struct knoten *node){
if (node!=0) {
    printf("\n Inhalt %s\n", node->wort);
    if(node->links!=NULL)
        bearb_pre(node->links);
    if(node->rechts!=NULL)
        bearb_pre(node->rechts);
    }
}
void bearb_post(struct knoten *node) {
if (node!=0) {

    if(node->links!=NULL) {
        bearb_post (node->links);
    }
    if(node->rechts!=NULL) {
        bearb_post (node->rechts);
    }
    printf("\n Inhalt %s\n", node->wort);
    }
}

void bearb_in(struct knoten *node) {
if (node!=0)
    {

    if(node->links!=NULL)
        bearb_in (node->links);
    printf("\n Inhalt %s\n", node->wort);
    if(node->rechts!=NULL)
        bearb_in (node->rechts);

    }
}
```

Grundlagen der Programmierung

Vorlesungsskript der Fachhochschule Osnabrück

Prof. Dr. T. Gervens, Prof. Dr.-Ing. B. Lang, Prof. Dr.-Ing. C. Westerkamp,
Prof. Dr.-Ing. J. Wübbelmann

16 Automatisches Übersetzen von Programmen mit *make*

16.1 Einleitung

Beim Übersetzen eines Programmes, welches aus mehreren Quelldateien besteht, muss bei Änderungen meist nur ein kleiner Teil der Quelldateien neu übersetzt werden. Die meisten Programmteile werden nicht modifiziert, liegen schon als gültige Objektdatei vor und brauchen somit nur noch mit den geänderten Objektdateien zusammengebunden werden. Dies spart schon bei mittelgroßen Programmen viel Zeit bei der Übersetzung.
Allgemeiner betrachtet möchte man eine Datei erzeugen, die von den Quelldateien (engl. source) direkt oder indirekt abhängig ist. Zwischen der gewünschten, abhängigen Datei (engl. target) und den zugehörigen Quelldateien lässt sich ein Abhängigkeitsgraph zeichnen.
Zum Erzeugen der abhängigen Datei aus den Quelldateien dienen Kommandos. Mit ihnen werden schrittweise alle abhängigen Dateien erzeugt, bis man schließlich die gewünschte Datei erzeugen kann.

Beispiel:

Ein Programm bestehe aus den Programmteilen *beispiel.c*, *teil2.c*, *teil3.c* und der Header-Datei *beispiel.h*. Die Header-Datei wird von jedem Programmteil mittels der Anweisung **`#include "beispiel.h"`** eingebunden.
Es soll die abhängige Datei *beispiel.exe* erzeugt werden. Diese lässt sich aus den abhängigen Dateien *beispiel.o, teil2.o,*und *teil3.o* mit dem g*cc*-Kommando erzeugen. Die drei Objekt-Dateien sind nun ihrerseits abhängig von den Quelldateien.
Sind alle Objektdateien *beispiel.o, teil2.o,*und *teil3.o* aktuell, muss bei einer Änderung von *teil2.c* nur die Objektdatei *teil2.o* neu erzeugt werden. Dies wird mit folgendem Kommando durchgeführt:

gcc –c teil2.c

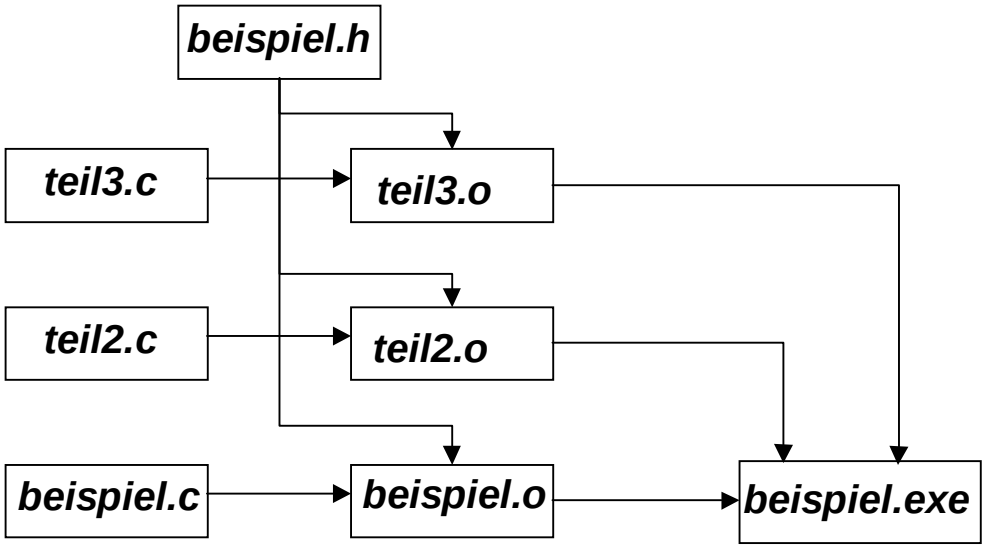
Anschließend müssen alle Objektdateien neu zusammengebunden werden, um die gewünschte ausführbare Datei *beispiel.exe* zu erzeugen. Dazu wird das folgende Kommando verwendet:

gcc beispiel.o teil2.o teil3.o

Wird hingegen die Header-Datei geändert, so müssen alle Objektdateien neu erzeugt werden:

gcc –c beispiel.c
gcc –c teil2.c
gcc –c teil3.c
gcc beispiel.o teil2.o teil3.o –o beispiel.exe

Es ergibt sich der folgende Abhängigkeitsgraph:



Zur Erzeugung der gewünschten Datei muss der Abhängigkeitsgraph durchsucht werden. Wenn Quelldateien aktueller sind als davon direkt oder indirekt abhängige Dateien, müssen diese neu erzeugt werden.
Zur Automatisierung dieses Vorgehens dient das *make*-Kommando. Es liest eine Spezifikationsdatei, in der die Abhängigkeiten zwischen den Dateien beschrieben und die Kommandos zur Erzeugung abhängiger Dateien spezifiziert sind. Üblicherweise wird diese Spezifikationsdatei als *Makefile* bezeichnet.

16.2 *Makefile*-Struktur

16.2.1 Abhängigkeitszeilen

In einem *Makefile* werden die direkten Abhängigkeiten eines Targets (abhängige Datei) von mehreren anderen Targets oder Quelldateien in Abhängigkeitszeilen beschrieben. Diese haben folgende Form:

<Target-Name>: <Liste der Abhängigkeiten>

Die Bezeichnung *<Target-Name>* steht dabei für den Namen der abhängigen Datei. Die *<Liste der Abhängigkeiten>* beinhaltet die Namen aller Dateien, die zur Erzeugung der abhängigen Datei benötigt werden.

Beispiel:

Die Abhängigkeitszeile für *beispiel.exe* (siehe oben) lautet:

beispiel.exe: beispiel.o teil2.o teil3.o

Die Abhängigkeitszeile für *teil2.o* lautet:

teil2.o: teil2.c beispiel.h

16.2.2 Regeln

Nach jeder Abhängigkeitszeile werden Regeln mit Kommandos angegeben, mit denen die abhängige Datei (Target) aus direkt übergeordneten Dateien erzeugt wird. In der *Makefile*-Syntax müssen diese Kommandozeilen mit einem Tabulatorzeichen beginnen.

Beispiel:

Die Abhängigkeitszeilen mit Kommandos zum Erzeugen für *beispiel.exe, beispiel.o, teil2.o* und *teil3.o* sind in folgender Datei *Makefile* gespeichert:

Makefile:

```
beispiel.exe: beispiel.o teil2.o teil3.o
<Tab>  gcc –o beispiel.exe beispiel.o teil2.o teil3.o
beispiel.o: beispiel.c beispiel.h
<Tab>  gcc –c beispiel.c
teil2.o: teil2.c beispiel.h
<Tab>  gcc –c teil2.c
teil3.o: teil3.c beispiel.h
<Tab>  gcc –c teil3.c
```

Die Bezeichnung <*Tab*> soll das Tabulatorzeichen anzeigen.

Wird das Kommando *make* aufgerufen, sucht es im aktuellen Verzeichnis nach der Datei *Makefile* und bildet aus den Abhängigkeitszeilen den Abhängigkeitsgraphen.
Beim Aufruf von *make* gibt man an, welches Target erzeugt werden soll. Entsprechend dem Abhängigkeitsgraphen überprüft *make*, welche abhängigen Dateien neu erzeugt werden müssen, um das gewünschte Target neu zu erzeugen. Dabei wird das Erstellungsdatum überprüft. Eine abhängige Datei muss dabei immer neuer sein als die zur Erzeugung notwendigen Dateien.
Wird beim Aufruf von *make* kein Target angegeben, so wird das erste in *Makefile* spezifizierte Target erzeugt.

Beispiel: Nach Änderung von *teil2.c* soll die ausführbare Datei *beispiel.exe* neu erzeugt werden. Dazu wird das Kommando *make* wie folgt aufgerufen:

make beispiel.exe

oder (weil beispiel.exe das erste Target in Makefile ist):

make

Im Abhängigkeitsgraphen wird festgestellt, dass zur Erzeugung von beispiel.exe die Dateien *beispiel.o, teil2.o* und *teil3.o* benötigt werden. Bei Überprüfung dieser Dateien ergibt sich, dass *beispiel.o,* neuer ist als die zugehörigen Quelldateien *beispiel.c, beispiel.h* und *teil3.o* entsprechend neuer ist als *teil3.c, beispiel.h*. Diese beiden abhängigen Dateien brauchen somit nicht neu erzeugt werden.
Die Überprüfung von *teil2.o* ergibt hingegen, dass die zugehörige Quelldatei *teil2.c* neuer ist, somit muss *teil2.o* neu erzeugt werden. Damit wird *teil2.o* neuer als *beispiel.exe*, somit muss anschließend auch *beispiel.exe* neu erzeugt werden.
make wendet nun die in den Regeln enthaltenen Kommandos in der durch den Abhängigkeitsgraphen vorgegebenen Reihenfolge an und ruft im Beispiel die folgenden Kommandos auf:

gcc –c teil2.c
gcc –o beispiel.exe beispiel.o teil2.o teil3.o

Eine Abhängigkeitszeile mit mehreren Abhängigkeiten auf der rechten Seite kann in mehrere Abhängigkeitszeilen aufgeteilt werden Das Kommando wird dann nur einer Abhängigkeitszeile zugeordnet.

16.2.3 Kommentare

Kommentarzeilen werden in einem *Makefile* mit dem Zeichen ‚*#*' eingeleitet. Alle Zeichen hinter dem ‚*#*'-Zeichen in der aktuellen Zeile werden als Kommentar interpretiert und von *make* nicht ausgewertet.

16.2.4 Folgezeilen

Soll der Inhalt einer logischen Zeile in mehreren Zeilen der Makefile-Datei dargestellt werden, so kann das ‚\'-Zeichen direkt am Zeilenende eingefügt werden. Die nachfolgende Dateizeile wird dann logisch an die vorhergehende Zeile angehängt.

Beispiel:

Die Regel:

beispiel.exe: beispiel.o teil2.o teil3.o
<Tab> gcc –o beispiel.exe beispiel.o teil2.o teil3.o

kann beispielsweise wie folgt geschrieben werden:

beispiel.exe: beispiel.o \<Zeilenende>
teil2.o\<Zeilenende>
teil3.o
<Tab> gcc –o beispiel.exe beispiel.o teil2.o teil3.o

Die Bezeichnung *<Tab>* soll dabei das Tabulatorzeichen und die Bezeichnung *<Zeilenende>* das oder die Zeilenendezeichen andeuten.

16.3 Variable

16.3.1 Definition von Variablen

In einer *Makefile*-Datei können Variable definiert und referenziert werden.
Die Definition einer Variablen erfolgt durch Spezifikation eines Variablennamens gefolgt von einem Gleichheitszeichen. Der Rest der Zeile hinter dem Gleichheitszeichen und dem Zeilenende wird zum Inhalt der Variable. Der Name einer Variable sollte aus Buchstaben, Ziffern und Unterstrichen bestehen (weitere Zeichen sind erlaubt, sollten aber vermieden werden).
Das Referenzieren einer Variablen erfolgt durch das ‚*$*'-Zeichen gefolgt von dem in runden Klammern eingeschlossenen Variablennamen.

Beispiel:

Die Regel:

```
# Tool-Definitionen
CC = gcc
# Datei-Definitionen
OBJECTS = beispiel.o teil2.o teil3.o
PRODUCT = beispiel.exe
# Regel zum Erzeugen des Produkts
#   durch Binden der Objektdateien
$(PRODUCT): $( OBJECTS)
        $(CC) –o $(PRODUCT) $(OBJECTS)
```

In obiger Definition werden die Variablen *CC*, *OBJECTS* und *PRODUCT* vereinbart.
Der Wert von beispielsweise *OBJECTS* ist der folgende String: „*beispiel.o teil2.o teil3.o*". Die Referenz *$(OBJECTS)* ersetzt das *make*-Kommando durch diesen String.

Innerhalb eines *Makefile* kann mittels Variablenreferenz auf Umgebungsvariable zugegriffen werden. Sofern im *Makefile* keine Variable gleichen Namens besteht, verwendet der *make*-Befehl den Wert der Umgebungsvariablen.

16.3.2 Automatische (vorbesetzte) Variable

In den Kommandos der Regeln kann auf den Namen des Targets und auf die Dateinamen, von denen ein Target abhängig ist, durch automatische Variable zugegriffen werden. Die wichtigsten automatischen Variablen werden durch die in folgender Tabelle aufgeführten Referenzen angesprochen:

$@	Name des Targets.
$*	Name des Targets ohne Erweiterung (suffix).
$<	Name der direkt zugehörigen Datei in der Abhängigkeitsliste.
$^	Alle Dateinamen der Abhängigkeitsliste.
$?	Alle Dateien der Abhängigkeitsliste, welche neuer als das Target sind.

Beispiel:

Das bereits vorgestellte *Makefile* zum Erzeugen von *beispiel.exe* kann mit automatischen Variablen wie folgt umgeschrieben werden:

Datei *Makefile:*

```
beispiel.exe: beispiel.o teil2.o teil3.o
        gcc –o $@ $^
beispiel.o: beispiel.c beispiel.h
        gcc –c $<
teil2.o: teil2.c beispiel.h
        gcc –c $<
teil3.o: teil3.c beispiel.h
        gcc –c $<
```

Man erkennt, dass durch Verwendung der automatischen Variablen alle Kommandos zum Übersetzen der C-Quelldateien in Objektdateien gleich geworden sind.

Beispiel:

Automatischen Variablen kann man beispielsweise mit einem *Makefile* untersuchen, welches Regeln mit dem *echo*-Kommando besitzt:

```
all.xxx: d1.aaa d2.bbb
        echo 1: "$*" "$@" "$<" >all.xxx
d1.aaa:
        echo 2: "$*" "$@" "$<" >d1.aaa
d2.bbb:
        echo 3: "$*" "$@" "$<" >d2.bbb
```

Durch Löschen einzelner mit dem *echo*-Kommando erzeugter Testdateien kann man weiterhin untersuchen, wie der Abhängigkeitsgraph abgearbeitet wird.

16.4 Pattern-Regeln

Bei Verwendung von Default-Regeln gibt man in einer *Makefile*-Datei für Standardaufgaben (wie z.B. das Übersetzen der C-Quellen in Objektdateien) im Normalfall keine Regel mehr an. Nur wenn besondere Optionen gewünscht werden, wird eine Regel explizit angegeben.
Defaultregeln basieren auf Dateiendungen (z.B. „.c“, „.o“, „.exe“).

Eine der wichtigsten Default Regel ist die *pattern rule:*
Hierzu wird eine Regel aufgestellt, die aus allen Dateien mit einer bestimmten Endung Dateien mit einer anderen Endung erzeugt.
Soll beispielsweise aus C Quellcode Dateien Objektdateien erzeugt werden, kann folgende Regel angegeben werden:

```
%.o: %.c
<Tab> $(CC) $(OPTIONS) $<
```

Der Name der Datei (ohne Erweiterung!) wird durch das % -Zeichen repräsentiert.
Wird in einer Regel eine Datei mit der Endung .o benutzt, so durchsucht make die pattern Regeln, ob es eine Regel zu Erzeugung der Objektdatei findet und generiert es, falls es die zughörige C Quelldatei findet. Es muss keine eigene Regel für jede Objektdatei angegeben werden.

Beispiel:

Das vorgestellte *Makefile* zum Erzeugen von *beispiel.exe* wird um Suffix-Regeln erweitert:

Datei *Makefile:*

```
# ------------------------------------------------------------
```

```
# Beispiel-Makefile zum Erzeugen der ausfuehrbaren
# Programmdatei beispiel.exe
# ------------------------------------------------------------
# Tools
CC = gcc
CFLAGS=-Wall -pedantic
# Dateien
OBJECTS = beispiel.o teil1.o teil2.o teil3.o
# Regel zum Binden
beispiel.exe: $(OBJECTS)
	$(CC) -o $@ $^
# Abhaengigkeitsregeln
beispiel.o: beispiel.c beispiel.h
teil1.o: teil1.c beispiel.h
teil2.o: teil2.c beispiel.h
teil3.o: teil3.c beispiel.h
# pattern Regel
%o : %.c
$(CC) $(CFLAGS) -c $<
```

16.5 Spezialitäten: .PHONY-Targets

Es ist guter *Makefile*-Stil, eine Regel vorzusehen, mit der alle erzeugten Dateien eines *Makefile* gelöscht werden können. Das Target dieser Regel erhält normalerweise den Namen *clean* und ist nicht von Dateien abhängig. Als Kommando der Regel werden Löschbefehle für die erzeugten Dateien angegeben.
Ein Problem ergibt sich, wenn sich eine Datei mit Namen *clean* im aktuellen Verzeichnis befindet. Da diese Datei laut Abhängigkeitszeile von keiner Datei abhängt, werden die zugehörigen Kommandos nie ausgeführt und die Dateien nie gelöscht.
Abhilfe schafft hier der Makefile-Befehl *.PHONY*. Er gibt an, dass das zugehörige Target immer ungültig ist. Dadurch werden die nachfolgenden Kommandos immer ausgeführt.

Beispiel:

Das *Makefile* zum Erzeugen von *beispiel.exe* erhält zusätzlich die clean-Regel zum Löschen aller erzeugten Dateien:

Datei *Makefile:*

```
# ------------------------------------------------------------
# Beispiel-Makefile zum Erzeugen der ausführbaren
# Programmdatei beispiel.exe
# ------------------------------------------------------------
# Tools
CC = gcc
RM = rm
... (Mittlerer Teil wie obiges Beispiel)
# Regel zum Löschen aller erzeugter Dateien
.PHONY : clean
clean:
        $(RM) beispiel.exe
        $(RM) $(OBJECTS)
```

16.6 Kommandozeilenparameter

Die wichtigsten Kommandozeilenparameter beim Aufruf von make sind in nachfolgender Tabelle aufgeführt:

-n	Die Kommandos, die zum Erzeugen des beim *make*-Befehl spezifizierten Targets ausgeführt werden müssen, werden nicht ausgeführt sondern nur angezeigt. Dies ist hilfreich zum Debuggen einer *Makefile*-Datei.
-d	Debuggen einer Makefile-Datei: Information ausdrucken, welche Targets neu erzeugt und welche Kommandos ausgeführt werden.
-f <Dateiname>	Statt der Datei mit Namen *Makefile* wird die spezifizierte Datei als *Makefile* verwendet.

Grundlagen der Programmierung

Vorlesungsskript der Fachhochschule Osnabrück

Prof. Dr.-Ing. B. Lang, Prof. Dr.-Ing. C. Westerkamp

17 Versionsverwaltung von Quelldateien

Bei der Entwicklung von Quelldateien ist es notwendig, die gemeinsame Entwicklung mehrerer Entwickler an einem Projekt zu koordinieren. Auch bei Projekten, die von einem Entwickler bearbeitet werden, ist es sinnvoll, auf frühere Entwicklungsstände zurückgreifen zu können.

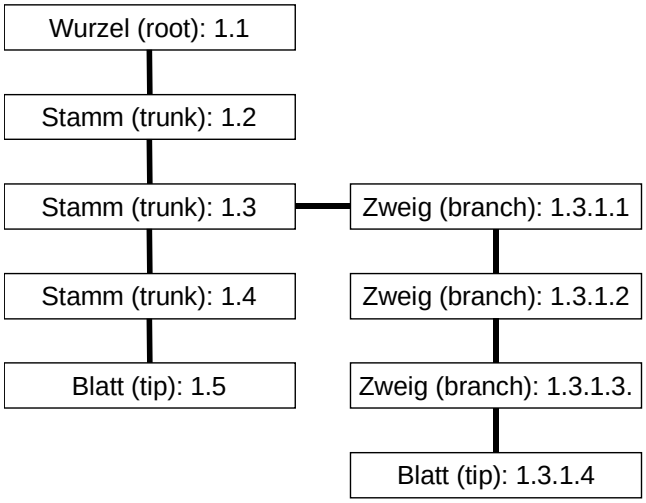
- Frühere Entwicklungsstände benötigt man beispielsweise, wenn Änderungen durchgeführt wurden, die sich als fehlerhaft herausstellten und die wieder rückgängig gemacht werden müssen.
- Bei Auslieferung eines Softwareproduktes ist es notwendig, den Stand der Quelldateien festzuhalten. Treten später Fehler in der Software auf, kann man diese nachvollziehen und beheben, auch wenn die Weiterentwicklung des Softwarepaketes schon deutlich fortgeschritten ist.
- Bei der Entwicklung von Programmpaketen mit mehreren Entwicklern muss der gemeinsame Zugriff auf die Dateien organisiert werden. Dateien dürfen nur von einem Entwickler verändert werden. Außerdem können parallele Änderungen an verschiedenen Quelldateien zusammengeführt werden.

Zur Unterstützung der genannten Aufgaben existieren vielfältige Programmpakete wie z.B. MS SourceSafe, Continuum, ClearCase, CVS (Open Source), ComponentSoftware RCS[1] und weitere. Diese Programmpakete ermöglichen neben der Versionsverwaltung einzelner Dateien meist auch die Verwaltung von Projekten, in denen zusammengehörigen Einzeldateien zusammengefasst sind. Die elementaren Kommandos zur Versionsverwaltung sollen am Beispiel **RCS** betrachtet werden. Es stellt die notwendigen Kommandos zur Versionskontrolle von Einzeldateien als Kommandozeilenbefehle zur Verfügung.

17.1 Revisionsbaum

Das Bild auf der nächsten Seite zeigt beispielhaft den Revisionsbaum einer Datei. Für jede in die Revisionsverwaltung aufgenommene Datei existiert ein solcher Revisionsbaum. Die Entwicklung beginnt bei Revision 1.1, diese Revision bildet bei der ersten Aufnahme einer Datei die Wurzel des Revisionsbaumes.

[1] Die Einzelplatzlizenz von ComponentSoftware RCS wird als Freeware zur Verfügung gestellt. Weitere Infos unter: http://www.ComponentSoftware.com/csrcs/news.htm



Wird die Quelldatei in Schritten verändert und zwischenzeitlich wieder in die Versionsverwaltung „eingecheckt“, wird bei jedem Ein-Checken eine neue Revisionsnummer vergeben. Man erkennt dies am Stamm des Revisionsbaumes, wo die Revisionen 1.2 bis 1.5 angelegt sind.

RCS erlaubt es, von Revisionen abzuzweigen, um von dort aus Seitenzweige der Datei zu bearbeiten. Im Bild erfolgt eine Verzweigung bei Revision 1.3. Es wird der Seitenzweig mit der Bezeichnung 1.3.1 angelegt und die erste Datei dieses Seitenzweiges mit Revision 1.3.1.1 bezeichnet.

17.2 Initialisierung des RCS-Paketes

Unter UNIX wird üblicherweise keine benutzerspezifische Initialisierung benötigt. Der Benutzername kann von den Programmen vom Betriebssystem erfragt werden. Bei Verwendung von RCS unter DOS oder im DOS-Kommandofenster von Windows muss üblicherweise der Benutzername mit:

set LOGNAME=<Benutzername>

gesetzt werden. Weiter müssen die RCS-Kommandodateien im ausführbaren Pfad stehen.

Im Verzeichnis, in dem die Quelldateien stehen, ist ein Unterverzeichnis „RCS“ einzurichten, in dem eine Datenbasis für jede Quelldatei eingerichtet wird. Unter UNIX hat diese Datenbasis den Namen der Quelldatei mit einer zusätzlichen Endung *",v"*. Wegen der Beschränkungen für Dateinamen unter DOS wird dort meist der unmodifizierte Name der Quelldatei für die Datenbasis verwendet.

17.3 Ein- und Aus-Checken einer Datei

Zum Hinzufügen einer neuen Datei zur Versionsverwaltung oder zum Speichern einer modifizierten Datei in der Datenbasis (Ein-Checken) dient das Kommando *ci*. In seiner einfachsten Form erwartet es den Dateinamen als Argument:

ci <Dateiname>

Beispiel:

ci beispiel.c

Bei neuen Dateien wird eine Beschreibung (description) der Datei, bei späteren Änderungen ein Kommentar zur veränderten Revision (log message) erfragt.

Beim Ein-Checken einer neuen Datei erhält man beispielsweise die folgende Aufforderung zur Eingabe einer Beschreibung:

```
>ci beispiel.c
RCS/beispiel.c  <--  beispiel.c
enter description, terminated with single '.' or end of file:
NOTE: This is NOT the log message!
>>
```

Beim erneuten Ein-Checken einer geänderten Datei erhält man folgende Aufforderung:

```
>ci beispiel.c
RCS/beispiel.c  <--  beispiel.c
new revision: 1.2; previous revision: 1.1
enter log message, terminated with single '.' or end of file:
>>
```

Der Kommentar bzw. die Beschreibung wird eingegeben und mit einem Punkt in einer sonst leeren Zeile beendet.

Nach dem Ein-Checken wird die Datei im Arbeitsverzeichnis gelöscht. Möchte man weiterhin eine Kopie der Datei im Arbeitsverzeichnis erhalten, kann man den Schalter *–u* beim Aufruf des ci-Kommandos spezifizieren:

ci –u <Dateiname>

Diese Kopie wird jedoch schreibgeschützt, um ein unkontrolliertes Ändern der Datei zu verhindern.

Man erhält die aktuellste Version des Hauptzweiges durch Aus-Checken mit dem *co*-Kommando. In der einfachsten Version geschieht dies mit Lesezugriff:

co <Dateiname>

Dadurch wird eine Kopie der aktuellen Version schreibgeschützt im Arbeitsverzeichnis angelegt.

Möchte man die Datei ändern, muss man sie zum Bearbeiten aus-checken. Dabei wird die Datenbasis gegen ein zweites Aus-Checken zum Bearbeiten gesperrt:

co –l <Dateiname>

Das *–l* steht für Lock (Verriegelung, Schloss). Beim Ein-Checken einer Datei mit dem *ci*-Kommando wird die Verriegelung entfernt, so dass ein anderer Bearbeiter die Datei aus-checken und bearbeiten kann. Möchte man einen

Zwischenstand ein-checken und die Verriegelung beibehalten, wendet man *ci* mit dem *–l*-Schalter an. Dadurch bleibt die Datei schreibbar im Arbeitsverzeichnis.

17.4 Einrichten eines Seitenzweiges

An bestimmten Stellen eines Projektes möchte man einen Zweig (Branch) einrichten, der unabhängig vom Hauptzweig weiterentwickelt wird. Dies geschieht durch Aus-Checken einer alten Revision zum Ändern:

co –l –r<Revision> <Dateiname>

Beim nachfolgenden Ein-Checken zeigt der *ci*-Befehl, dass ein Seitenzweig angelegt worden ist:

```
>ci beispiel.c
RCS/beispiel.c  <--  beispiel.c
new revision: 1.2.1.1; previous revision: 1.2
enter log message, terminated with single '.' or end of file:
>>
```

Will man weiter auf diesem Seitenzweig die Daten bearbeiten, genügt es beim Aus-Checken die Bezeichnung des Seitenzweigs (im Beispiel: *1.2.1*) anzugeben. Dann wird die aktuellste Revision dieses Seitenzweiges ausgecheckt:

co –l –r1.2.1 beispiel.c

Beim nachfolgenden Ein-Checken wird eine neue Revision des Seitenzweiges angelegt.

Es gibt weitere Befehle zum Ändern der Revisionsnummern.

Es ist möglich, beim Ein-Checken mit der *–r* Option eine neue Revisionsnummer abweichend von der vorgegebenen Sequenz zu spezifizieren:

ci –r<Revision> dateiname

Wurde beispielsweise die Entwicklung einer Datei abgeschlossen und man möchte von einer Revision *1.x* auf eine Revision *2.x* wechseln, so kann man dies beim Ein-Checken mit:

ci –f –r2 beispiel.c

erreichen. Die Option *–f* dient dazu, die Datei auch dann mit der neuen Revision einzuchecken, wenn sie nicht geändert wurde. Normalerweise erkennt RCS eine unveränderte Datei beim Ein-Checken und behält die vorhandene Revision bei.

17.5 Verwaltung und Änderungen der Datenbasis

Zum Überprüfen der Revisionsinformation einer Datei dient das *rlog*-Kommando. Es listet die in der Datenbasis vorhandenen Revisionen auf:

rlog <Dateiname>

Möchte man den Lock einer Datei in der Datenbasis explizit setzen, kann man dies mit dem *rcs*-Befehl erreichen:

rcs –l <Dateiname>

Achtung: Bei diesem Vorgehen wird die Datei nicht ausgecheckt, sondern nur ein Lock in der Datenbasis auf die aktuelle Revision gesetzt. Legt man eine Datei gleichen Namens an, kann diese mit dem *ci*-Befehl in die Datenbasis eingecheckt werden, so als hätte man sie vorher mit *co* ausgecheckt.

Entsprechend kann nach dem Aus-Checken einer Datei der Lock mit dem folgenden *rcs*-Befehl in der Datenbasis zurückgesetzt werden:

rcs –u <Dateiname>

Dann bleibt die Datei beschreibbar im aktuellen Verzeichnis erhalten. Ein nachfolgendes Ein-Checken führt jedoch zu einem Fehler.

Das folgende Beispiel zeigt das Setzen und nachfolgende Löschen des Locks in der Datenbasis der Datei *beispiel.c*:

>rcs -l beispiel.c
RCS file: RCS/beispiel.c
2.1 locked
done
>rcs -u beispiel.c
RCS file: RCS/beispiel.c
2.1 unlocked
done
>

17.6 Revisionsnamen

Bei der Auslieferung eines Programms ist es sinnvoll, alle aktuellen Revisionen der Quelldateien mit einem Namen zu versehen (Labeln). Dazu sollte man vorher alle Quelldateien ein-checken. Der an dieser Stelle vergebene Name wird auch als „Label“ bezeichnet.

Die Vergabe eines Namens an eine Revision einer Datei erfolgt mit dem *rcs*-Befehl und der *–l*-Option:

rcs –n<Name>:<Revision> <Dateiname>

Der nachfolgende Befehl weist beispielsweise der Revision *1.4* der Datei *beispiel.c* den Namen *BETATEST* in der zugehörigen Datenbasis zu:

rcs –nBETATEST:1.4 beispiel.c

Alternativ kann direkt beim Ein-Checken mit dem *ci*-Kommando mit der *–n*-Option ein Label-Name angegeben werden. Dann wird keine Revision angegeben:

ci –n<Name> <Dateiname>

Dann erhält die beim Ein-Checken neu angelegte Revision der Datei den spezifizierten Label-Namen.

Beim Aus-Checken der Datei kann statt der Revisionsnummer der vergebene Name in der *–r*-Option angegeben werden:

co –l –r<Name> <Dateiname>

Lässt man beim *rcs*-Kommando bei der *–n*-Option die Bezeichnung der Revision weg, wird der angegebene Name in der Datenbasis gelöscht:

rcs –n<Name> <Dateiname>

Dies ist beispielsweise notwendig, wenn nach Vergabe des Namens doch noch eine Änderung durchgeführt werden muss und der vergebene Name der neu erstellen Revision zugeordnet werden muss.

Nachfolgendes Beispiel zeigt, wie in der Datei *beispiel.c* eine neue Revision mit dem bereits vergebenen Label-Namen *BETATEST* versehen wird.

co –rBETATEST beispiel.c	Datei mittels Label-Namen aus-checken
rcs –nBETATEST beispiel.c	Label-Namen in Datenbasis löschen
edit beispiel.c	Datei editieren und verändern
ci –nBETATEST beispiel.c	Datei Ein-Checken und Label-Namen neu vergeben

Das Verschieben des Label-Namens ist alternativ ohne Löschen mit der *–N*-Option des *rcs*-Kommandos möglich:

rcs –N<Name>:<Revision> <Dateiname>

17.7 RCS und make

Die einfache Einbindung der wichtigsten RCS-Kommandos in eine *Makefile*-Datei soll das nachfolgende Listing zeigen. Das gelistete *Makefile* erzeugt die ausführbare Datei *beispiel.exe* aus den 4 Quelldateien *beispiel.c*, *teil2.c*, *teil3.c* und *beispiel.h*. Das gezeigte *Makefile* wird wie die Source-Dateien in die Versionsverwaltung verwaltet.

```
# ------------------------------------------------------
# Beispiel-Makefile zum Erzeugen der ausführbaren
# Programmdatei beispiel.exe (Einbindung von RCS)
# ------------------------------------------------------
# suffix-Regeln
.SUFFIXES:
.SUFFIXES: .c .o .exe .h
# Default-Regel zum Compilieren
.c.o:
     $(CC)  $(CFLAGS) -c $<
# Default-Regel zum Aus-Checken der Quellen (ohne Lock)
.c:
     co -u $@
.h:
```

```
	co -u $@
# tools
CC = gcc
# dateien
OBJECTS  = beispiel.o teil2.o teil3.o
SOURCES  = beispiel.c teil2.c teil3.c Makefile beispiel.h
LABEL    = VERSION_1_2
REVISION = 1
# Regel zum Binden
beispiel.exe: $(OBJECTS)
	$(CC) -o $@ $^
# Abhängigkeitsregeln
beispiel.o: beispiel.c beispiel.h
teil2.o: teil2.c beispiel.h
teil3.o: teil3.c beispiel.h
# Regel zum Löschen aller erzeugter Dateien
.PHONY : clean
clean: checkin
	rm beispiel.exe
	rm $(OBJECTS)
#---------------------------------------------------------
# RCS
#---------------------------------------------------------
# Aus-Checken der Dateien zum Bearbeiten (mit Lock)
.PHONY : checkout
checkout:
	co -l $(SOURCES)
# Aus-Checken der Dateien zum Bearbeiten mittels Label-
Namen
# (mit Lock)
.PHONY : checkout_label
checkout_label:
	co -l -r$(LABEL) $(SOURCES)
# Ein-Checken der Dateien nach Bearbeiten (Lock freigeben)
.PHONY : checkin
checkin:
	-ci $(SOURCES)
	co -u Makefile
# Vergabe eines Label-Namens. Evtl. ausgecheckte Dateien
# werden vorher eingecheckt
.PHONY : set_label
set_label: checkin
	rcs -n$(LABEL) $(SOURCES)
	co -l $(SOURCES)
	ci -u -n$(LABEL) $(SOURCES)
# Logging Information ausgeben
.PHONY : rlog
rlog:
	rlog $(SOURCES)
```

Interessant sind die Default-Regeln zum Aus-Checken der *.c* und *.h*-Dateien. Dort ist nur ein Target-Suffix spezifiziert. Somit wird das Target immer erzeugt, wenn keine entsprechende Datei vorliegt.

Zur Bearbeitung werden alle Quelldateien über die `checkout`-Regel ausgecheckt. Dort wird beim *co*-Kommando die *–l*-Option gesetzt.

Nach einer Bearbeitung erfolgt das Ein-Checken mit der `checkin`-Regel. Zu beachten ist, dass nach dem Ein-Checken die *Makefile*-Datei mit der *–u*-Option, d.h. ohne Lock wieder ausgecheckt wird. Damit ist das weitere Arbeiten mit dem *make*-Kommando gewährleistet. Das Minus-Zeichen vor dem *ci*-Kommando bewirkt, dass *make* die Bearbeitung seiner Kommandos fortsetzt, auch wenn der *ci*-Befehl einen Fehler meldet. Dies ermöglicht den Aufruf der `checkin`-Regel ohne Abbruch des *make*-Kommandos, auch wenn die Quelldateien bereits eingecheckt sind.

Die `checkin`-Regel wird durch die `clean`-Regel aufgerufen. Dadurch werden beim Aufruf von `clean` alle Quelldateien in die zugehörigen Datenbasis eingeschrieben und die Dateien im Arbeitsverzeichnis gelöscht (außer dem *Makefile*, siehe oben).

Beim Aufruf der `set_label`-Regel werden zunächst die Quelldateien über die `checkin`-Regel eingecheckt. Dann wird der Label-Name in der Datenbasis jeder Quelldatei gelöscht. Anschließend werden die Dateien ausgecheckt, so dass mit dem *ci*-Kommando der Label-Name an die aktuellen Revisionen der Dateien ohne Spezifikation der Revisionnummern vergeben werden kann.

Die Vergabe eines neuen Labels erfolgt in den folgenden Schritten:

1. Aus-Checken der Quelldateien.
2. Vergabe eines neuen Label-Namens in der *Makefile*-Datei (Variable `LABEL`)
3. Ein-Checken der Quelldateien und Setzen des neuen Labels mit der `set_label`-Regel.