Informatik I

— Grundlagen der Informatik

Fachhochschule Niederrhein

WS 2005/06

Prof. Dr. Rethmann

Literatur

Informatik allgemein

- Rechenberg: Was ist Informatik?
  Carl Hanser Verlag
- Gumm, Sommer: Einführung in die Informatik
  Oldenbourg Verlag

Programmiersprache C

- Kernighan, Ritchie: Programmieren in C
  Carl Hanser Verlag
- Zeiner: Programmieren lernen mit C
  Carl Hanser Verlag

Literatur (2)

Algorithmen

- Wirth: Algorithmen und Datenstrukturen
  Teubner Verlag
- Sedgewick: Algorithms, Addison-Wesley
- Ottmann, Widmayer: Algorithmen und Datenstrukturen
  BI Wissenschaftsverlag
- Cormen, Leiserson, Rivest: Introduction to Algorithms
  MIT Press
- Aho, Hopcroft, Ullman: Datastructures and Algorithms
  Addison-Wesley

Einleitung

Informatik

Was ist das?

- Kunstwort aus **Infor**mation und Mathe**matik**
- Informatik ist eng mit Computern verknüpft: solange es keine Computer gab, gab es auch keine Informatik
- elektronische Rechenmaschine entstand um 1940

Ursprung

- Rechnen galt bis Anfang der Neuzeit als **Kunst**
- heute kann jeder die vier Grundrechenarten ausführen
  * mechanisch ausführbares Verfahren, dass nicht verstanden werden muss, um es anwenden zu können
  * kann einer Maschine übertragen werden kann

Informatik (2)

Algorithmus

- mechanisch ausführbares Rechenverfahren
- bildet den Kern der Informatik
- nach dem persischen Mathematiker Al-Chowarizmi

Beispiel: Euklidischer Algorithmus

- berechne größten gemeinsamen Teiler zweier natürlicher Zahlen p und q
- Euklid lebte etwa 300 v.Chr.

Informatik (3)

Euklidischer Algorithmus

1. Man dividiere p ganzzahlig durch q. Dabei erhält man den Rest r, der zwischen 0 und $q-1$ liegt.
2. Wenn $r = 0$ ist, dann ist q der ggT. Wenn $r \neq 0$ ist, dann benenne das bisherige q in p um, das bisherige r in q und wiederhole ab Schritt 1.

p	q	$r = p \bmod q$
216	378	216
378	216	162
216	162	54
162	54	0

Informatik (4)

```
#include <stdio.h>
int main(void) {
    int a, b, p, q, r;
    scanf("%d, %d", &a, &b);
    p = a;
    q = b;
    r = p % q;
    while (r != 0) {
        p = q;
        q = r;
        r = p % q;
    }
    printf("ggT(%d,%d) = %d\n", a, b, q);
    return 0;
}
```

Informatik (5)

- Mittels `#include <stdio.h>` wird eine Bibliothek bereitgestellt, die Funktionen zur Ein- und Ausgabe enthält.
- Der Start eines Programms besteht im Ausführen der Funktion `main`.
- `int a` deklariert eine Variable `a`, die ganze Zahlen aufnehmen kann. Alle in einem C-Programm benutzten Variablen müssen explizit deklariert werden, wobei der Typ und der Name der Variablen festgelegt werden.
- Die Funktion `scanf()` liest Werte von der Tastatur ein, `printf()` gibt eine Zeichenkette auf dem Bildschirm aus. Solche Standardfunktionen sind übersetzte Funktionen, die zur C-Implementierung gehören.
- Alle Anweisungen werden mit einem Semikolon beendet.

Informatik (6)

- Anweisungsfolgen werden mit geschweiften Klammern zusammengefasst, der geklammerte Block gilt als eine Anweisung.
- Mittels `while()` kann eine Anweisung mehrmals durchlaufen werden. Ist die gegebene Bedingung nicht erfüllt, wird die Schleife verlassen. Die Bedingung wird vor der ersten und nach jeder Ausführung der Anweisung geprüft.

Informatik (7)

Algorithmus

- mechanisches Verfahren, das aus mehreren Schritten besteht
- Schritte werden sequentiell ausgeführt, bis das Ergebnis gefunden ist (es gibt auch parallele Algorithmen)
- einzelne Abschnitte des Verfahrens können mehrfach durchlaufen werden (Iteration, Schleife)

Entwurf von Algorithmen

- finde eine Problemlösung
- formuliere sie in kleinen, elementaren Schritten

Informatik (8)

Es gibt sehr alte, immer noch aktuelle Algorithmen:

- je zwei natürliche Zahlen haben einen ggT $\rightarrow$ Euklid
- eine Matrix ist invertierbar $\Longleftrightarrow$ die Determinante ist ungleich Null $\rightarrow$ Gaußsches Eliminationsverfahren
- erst der Computer ermöglicht es, auch komplizierte Algorithmen mit tausenden von Schritten auszuführen

Technische Informatik

Aufbau und Konstruktion von Computern.

- Rechnerarchitektur
- Rechnerhardware
- Mikroprozessortechnik
- Rechnernetze

Praktische Informatik

Entwicklung und Erweiterung der Rechnereigenschaften. Programmierung und Nutzung von Computern.

- Betriebssysteme
- Benutzerschnittstellen
- Informationssysteme (Datenbanken)
- Programmiersprachen und Übersetzer
- Softwaretechnologie

Theoretische Informatik

Formale mathematische Grundlagen.

- Formale Sprachen
- Automatentheorie
- Berechenbarkeit
- Komplexitätstheorie
- Algorithmen & Datenstrukturen

Angewandte Informatik

Lösen spezieller Probleme in Anwendungsbereichen mittels Computer. Der Rechner wird als Werkzeug eingesetzt.

- Computergrafik
- Digitale Signalverarbeitung (Bild-/Spracherkennung)
- Simulation und Modellierung
- Künstliche Intelligenz
- Textverarbeitung

Anmerkungen

Praktische und Angewandte Informatik sind mitunter nur schwer abzugrenzen, weil in beiden die Programmierung im Mittelpunkt steht.

In letzter Zeit wird dies durch eine andere Art der Einteilung ergänzt: Wirtschafts-, Bio-, Geoinformatik usw.

Informatik ist nicht gleichzusetzen mit Programmierung.

Man lernt Informatik nicht aus Büchern wie

- „Word 7.0 für Fortgeschrittene“ oder
- „Die besten Tipps zum Surfen im Internet“ oder
- „Programmieren in C++“.

Inhalte der Vorlesung

- Zahlendarstellung im Rechner
  * Darstellung ganzer Zahlen
  * Darstellung von Gleitkommazahlen
  * Rechnerarithmetik
- Die Programmiersprache C
  * Top-down Entwicklung
  * Grundelemente der Sprache
  * Strukturierte Programmierung
  * Standardbibliotheken
  * Modulare Programmierung

Inhalte der Vorlesung (2)

- Algorithmen und Datenstrukturen
  * Aufwandsabschätzungen
  * Sortieralgorithmen
  * Suchalgorithmen
  * Graphalgorithmen
  * Hash-Verfahren
- Diverses
  * Formale Sprachen
  * Programmiersprachen
  * Modellbildung und Spezifikation
  * Rechnerarchitektur

Zahlendarstellung im Rechner

Allgemeines

Alphabet: In Daten vorkommende Zeichen gehören immer einer bestimmten Zeichenmenge an:

- Zahlen → Ziffern, Dezimalpunkt und Vorzeichen
- Texte → Buchstaben, Ziffern und Satzzeichen

Die einzelnen Zeichen werden **Symbole** genannt.

Das klassische Alphabet der indogermanischen Kultur ist

$$\Sigma_{10} = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$$

Wörter mit fester Länge werden durch Einfügen führender Nullen erreicht: 0123, 0974, 0007.

Allgemeines (2)

Weitere Alphabete:

Σ_2	=	$\{0, 1\}$	dual, binär
Σ_8	=	$\{0, 1, 2, 3, \ldots, 7\}$	oktal
Σ_{16}	=	$\{0, 1, 2, 3, \ldots, 9, A, B, C, D, E, F\}$	hexadezimal

Σ_{16} ist strenggenommen das Alphabet $\{0, 1, 2, \ldots, 14, 15\}$. Anstelle der „Ziffern“ $10, 11, \ldots$ werden generell neue Symbole $A, B, \ldots$ verwendet.

In der Informatik häufig zu finden: Basen $b = 2, 8, 16$

Von geringer Bedeutung: Basen $b = 12$ (Dutzend, Gros), $b = 20$ (franz. vingt) und $b = 60$ (Zeitrechnung).

Codierung

Die Symbole aller denkbaren Alphabete lassen sich durch **Gruppen von Binärzeichen** ausdrücken.

Beispiel: Das deutsche Alphabet der Kleinbuchstaben kann wie folgt dargestellt werden:

00000	a	00100	e	01000	i
00001	b	00101	f	01001	j
00010	c	00110	g	01010	k
00011	d	00111	h	01011	l ...

Wichtig: Mit Gruppen aus n Binärzeichen lassen sich 2^n Symbole codieren.

Codierung (2)

Ein **Code** ist die Zuordnung einer Menge von Zeichenfolgen zu einer anderen Menge von Zeichenfolgen.

Die Zuordnung (besser Abbildung) erfolgt oft durch eine Tabelle, der **Codetabelle**.

Ziffern, Klein- und Großbuchstaben, Umlaute, Satzzeichen und einige mathematische Zeichen können mit 8 Binärzeichen codiert werden.

International: **ASCII-Code** (American Standard Code for Information Interchange)

ASCII-Tabelle (Auszug)

Oct	Dec	Hex	Char	Oct	Dec	Hex	Char
054	44	2C	,	070	56	38	8
055	45	2D	-	071	57	39	9
056	46	2E	.	072	58	3A	:
057	47	2F	/	073	59	3B	;
060	48	30	0	074	60	3C	<
061	49	31	1	075	61	3D	=
062	50	32	2	076	62	3E	>
063	51	33	3	077	63	3F	?
064	52	34	4	100	64	40	@
065	53	35	5	101	65	41	A
066	54	36	6	102	66	42	B
067	55	37	7	103	67	43	C

b-adische Darstellung natürlicher Zahlen

Satz: Sei $b \in \mathbb{N}$ und $b > 1$. Dann ist jede ganze Zahl z mit $0 \leq z \leq b^n - 1$ und $n \in \mathbb{N}$ eindeutig als Wort $z_{n-1}z_{n-2}\ldots z_0$ der Länge n über Σ_b dargestellt durch:

$$z = z_{n-1} \cdot b^{n-1} + z_{n-2} \cdot b^{n-2} + \ldots + z_1 \cdot b^1 + z_0 \cdot b^0$$

Vereinbarungen:

- In der Ziffernschreibweise geben wir in der Regel die Basis der Zahlendarstellung explizit an, außer wenn eindeutig klar ist, welche Basis gemeint ist.
- Die Basis selbst wird immer dezimal angegeben.

b-adische Darstellung natürlicher Zahlen (2)

Beispiel: Sei $b = 10$ (Dezimalsystem).
Die eindeutige Darstellung von $z = 4711$ lautet

$$\begin{aligned} z &= 4 \cdot 10^3 + 7 \cdot 10^2 + 1 \cdot 10^1 + 1 \cdot 10^0 \\ &= 4 \cdot 1000 + 7 \cdot 100 + 1 \cdot 10 + 1 \cdot 1 \\ &= 4000 + 700 + 10 + 1 \end{aligned}$$

und in Ziffernschreibweise $(4711)_{10}$.

Beispiel: Sei $b = 2$ (Dualsystem).
Die eindeutige Darstellung von $z = 42$ lautet

$$\begin{aligned} z &= 1 \cdot 2^5 + 0 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0 \\ &= 1 \cdot 32 + 1 \cdot 8 + 1 \cdot 2 \end{aligned}$$

und in Ziffernschreibweise $(101010)_2$.

Darstellung reeller Zahlen

Festpunktdarstellung: Das Komma wird an beliebiger, aber fester Stelle angenommen.

Satz: Sei $b \in \mathbb{N}$ und $x_{n-1}x_{n-2}\ldots x_0 x_{-1}x_{-2}\ldots x_{-m}$ eine $n+m$-stellige Zahl mit $x_i \in \Sigma_b$ für $i = -m, -m+1, \ldots, n$, wobei das Komma rechts von x_0 angenommen wird. Dann stellt obiges Wort folgende Zahl dar:

$$\begin{aligned} x = {} & x_{n-1} \cdot b^{n-1} + x_{n-2} \cdot b^{n-2} + \ldots + x_1 \cdot b^1 + x_0 \cdot b^0 \\ & + x_{-1} \cdot b^{-1} + \ldots + x_{-m} \cdot b^{-m} \end{aligned}$$

Beispiel: 000010111011 ist bei 8 Vor- und 4 Nachkommastellen die Darstellung von

$$\begin{array}{ccccccccccccc} 2^3 & + & 2^1 & + & 2^0 & + & 2^{-1} & + & 2^{-3} & + & 2^{-4} & = \\ 8 & + & 2 & + & 1 & + & \frac{1}{2} & + & \frac{1}{8} & + & \frac{1}{16} & = & 11.6875 \end{array}$$

Darstellung reeller Zahlen (2)

Frage: Zahlen sind Zeichenketten. Warum codiert man Zahlen nicht im ASCII?

Antwort:

- **hoher Speicherbedarf:** Jede Ziffer benötigt 8 Zeichen zur Darstellung. **Beispiel:** Darstellung von $(123)_{10}$

  ASCII: 00110001 00110010 00110011
  Dual: 01111011

- **komplizierte Arithmetik:** ASCII-Werte können nicht einfach summiert werden!

8	00111000	$\widehat{=}$	8
+ 9	+ 00111001	$\widehat{=}$	9
17	01110001	$\widehat{=}$	q

Zahlenumwandlung
Dezimalsystem → andere Systeme

Beispiel: $(935.421875)_{10} = (3A7.6C)_{16}$.

1. Zahl aufteilen in Vor- und Nachkommateil.
2. Vorkommateil durch fortgesetzte Division umwandeln.

$$\begin{aligned} 935 &: 16 = 58 \text{ Rest } 7 \;\widehat{=}\; 7 \\ 58 &: 16 = 3 \text{ Rest } 10 \;\widehat{=}\; A \\ 3 &: 16 = 0 \text{ Rest } 3 \;\widehat{=}\; 3 \end{aligned}$$

Die jeweiligen Divisionsreste ergeben von unten nach oben gelesen den Vorkommateil der gesuchten Zahl in der anderen Basis.

Zahlenumwandlung
Dezimalsystem → andere Systeme (2)

3. Nachkommateil durch fortgesetzte Multiplikation umwandeln.

$$\begin{aligned} 0.421875 \cdot 16 &= 6 + 0.75 \to 6 \\ 0.75 \cdot 16 &= 12 + 0 \to C \end{aligned}$$

Die jeweiligen ganzen Teile ergeben von oben nach unten gelesen den Nachkommateil der gesuchten Zahl in der anderen Basis.

Korrektheit des Verfahrens: ...

Zahlenumwandlung
Dezimalsystem → andere Systeme (3)

Beispiel: $(978.127685546875)_{10} = (3D2.20B)_{16}$
Vorkommateil:

$$\begin{aligned} 978 &: 16 = 61 \text{ Rest } 2 \;\widehat{=}\; 2 \\ 61 &: 16 = 3 \text{ Rest } 13 \;\widehat{=}\; D \\ 3 &: 16 = 0 \text{ Rest } 3 \;\widehat{=}\; 3 \end{aligned}$$

Nachkommateil:

$$\begin{aligned} 0.127685546875 \cdot 16 &= 2 + 0.04296875 \to 2 \\ 0.04296875 \cdot 16 &= 0 + 0.6875 \to 0 \\ 0.6875 \cdot 16 &= 11 + 0 \to B \end{aligned}$$

Zahlenumwandlung
Dezimalsystem → andere Systeme (4)

Beispiel: $(122.1)_{10} = (172.0\overline{6314})_8$
Vorkommateil:

$$\begin{aligned} 122 &: 8 = 15 \text{ Rest } 2 \\ 15 &: 8 = 1 \text{ Rest } 7 \\ 1 &: 8 = 0 \text{ Rest } 1 \end{aligned}$$

Nachkommateil:

$$\begin{aligned} 0.1 \cdot 8 &= 0 + 0.8 \\ 0.8 \cdot 8 &= 6 + 0.4 \\ 0.4 \cdot 8 &= 3 + 0.2 \\ 0.2 \cdot 8 &= 1 + 0.6 \\ 0.6 \cdot 8 &= 4 + 0.8 \ \ldots \end{aligned}$$

Periodische Dualbrüche

Bei der Umwandlung vom Dezimal- ins Dualsystem ergeben sich oft periodische Dualbrüche: $(0.1)_{10} = (0.0\overline{0011})_2$.

Im Rechner:

- Länge der Zahlen ist beschränkt → periodische Dualbrüche können nur näherungsweise dargestellt werden
- bei n Nachkommastellen ist der durch das Weglassen weiterer Dualstellen entstehende Fehler im Mittel die Hälfte der letzten dargestellten Ziffer → $0.5 \cdot 2^{-n}$

Zahlenumwandlung
beliebige Systeme → Dezimalsystem

Berechnen der b-adischen Darstellung: **Horner-Schema**

Beispiel: $(63D2)_{16} = (25554)_{10}$

$$\begin{aligned} (63D2)_{16} &= 6 \cdot 16^3 + 3 \cdot 16^2 + 13 \cdot 16^1 + 2 \cdot 16^0 \\ &= (6 \cdot 16 + 3) \cdot 16^2 + 13 \cdot 16^1 + 2 \cdot 16^0 \\ &= [(6 \cdot 16 + 3) \cdot 16 + 13] \cdot 16 + 2 \end{aligned}$$

Durch Anwenden des Horner-Algorithmus reduziert sich die Anzahl der durchzuführenden Multiplikationen.

Beispiel: $(1736)_8 = (990)_{10}$

$$\begin{aligned} (1736)_8 &= 1 \cdot 8^3 + 7 \cdot 8^2 + 3 \cdot 8^1 + 6 \cdot 8^0 \\ &= [(1 \cdot 8 + 7) \cdot 8 + 3] \cdot 8 + 6 \end{aligned}$$

Umwandlung artverwandter Systeme

Bei zwei Basen $b, b' \in \mathbb{N}$ mit $b' = b^n$ für ein $n \in \mathbb{N}$ kann die Zahlenumwandlung vereinfacht werden.

Beispiel: $(21121,1)_3 = (247,3)_9$
Die Ziffern der Zahl $(21121,1)_3$ werden von rechts nach links paarweise zusammengefasst, da $9 = 3^2$:

$$\begin{aligned} (02)_3 &= (2)_9 \\ (11)_3 &= (4)_9 \\ (21)_3 &= (7)_9 \\ (10)_3 &= (3)_9 \end{aligned}$$

Umwandlung artverwandter Systeme (2)

Beispiel: $(32132)_4 = (39E)_{16}$
Die Ziffern der Zahl $(32132)_4$ werden von rechts nach links paarweise zusammengefasst, da $16 = 4^2$.

$$\begin{aligned}(03)_4 &= (3)_{16}\\ (21)_4 &= (9)_{16}\\ (32)_4 &= (E)_{16}\end{aligned}$$

Beispiel: $(2A7)_{16} = (0010\ 1010\ 0111)_2$
Die Ziffern der Zahl $(2A7)_{16}$ werden jeweils als 4-stellige Dualzahl geschrieben, da $16 = 2^4$:

$$\begin{aligned}(2)_{16} &= (0010)_2\\ (A)_{16} &= (1010)_2\\ (7)_{16} &= (0111)_2\end{aligned}$$

Arithmetik: Addition

Addition einstelliger Dualzahlen:

0	0	1	1
+ 0	+ 1	+ 0	+ 1
0	1	1	10

Beispiel: Addition mehrstelliger Zahlen

Dezimal		Dual	
	37		00100101
	+49		+00110001
	86		01010110

Arithmetik: Multiplikation

Multiplikation einstelliger Dualzahlen:

$$\begin{aligned}0 * 0 &= 0\\ 1 * 0 &= 0\\ 0 * 1 &= 0\\ 1 * 1 &= 1\end{aligned}$$

Beispiel: Multiplikation mehrstelliger Zahlen

Dezimal	$37 \cdot 21$
	37
	740
	777

Dual	$00100101 \cdot 00010101$
	100101
	10010100
	1001010000
	1100001001

Arithmetik: Division

Dezimal

5	0	4	:	42	= 12
4	2				
	8	4			
	8	4			
		0			

Dual

1	1	1	1	1	1	0	0	0	: 101010	= 1100
1	0	1	0	1	0					
	1	0	1	0	1	0				
	1	0	1	0	1	0				
						0				

Zahlendarstellung im Rechner
Codierung ganzer Zahlen

- Vorzeichen und Betrag
- Einer-Komplement
- Zweier-Komplement
- Exzess-Darstellung

Vorzeichen- und Betragdarstellung

Voraussetzung: feste Wortlänge

ganz links stehendes Bit: Vorzeichen ($0/1 \mathrel{\widehat{=}} +/-$)
restliche Bits: Darstellung des Betrags

Beispiel: Bei einer Wortlänge von 16 Bit können Zahlen zwischen $+2^{15}$ und -2^{15} dargestellt werden.

$$+92 = 0000000001011100$$
$$-92 = 1000000001011100$$

Nachteile:

- zwei Nullen: -0 und +0
- Addierer und Subtrahierer nötig
- Entscheidungslogik nötig: addieren oder subtrahieren?

Vorzeichen- und Betragdarstellung (2)

Entscheidungslogik: vier Fälle sind zu unterscheiden:

Operanden	Operation	Beispiel
$+x, +y$	$x + y$	5 + 2 = 5 + 2
$-x, -y$	$-(x + y)$	-5 - 2 = -(5 + 2)
$+x, -y$ mit $\|x\| \geq \|y\|$	$x - y$	5 - 2 = 5 - 2
$-x, +y$ mit $\|y\| \geq \|x\|$	$y - x$	-2 + 5 = 5 - 2
$+x, -y$ mit $\|x\| < \|y\|$	$-(y - x)$	2 - 5 = -(5-2)
$-x, +y$ mit $\|y\| < \|x\|$	$-(x - y)$	-5 + 2 = -(5-2)

Wir wollen die Subtraktion vermeiden, indem wir die Subtraktion auf die Addition zurückführen.

Komplementdarstellung

Sei $x = x_{n-1}x_{n-2}\ldots x_0$ eine n-stellige Dualzahl. Sei

$$\overline{x_i} = \begin{cases} 1 & \text{falls } x_i = 0 \\ 0 & \text{falls } x_i = 1 \end{cases}$$

Einer-Komplement: komplementiere die einzelnen Ziffern der Dualzahl:

$$\overline{x}^1 = \overline{x_{n-1}}\ \overline{x_{n-2}}\ldots\overline{x_0}$$

Zweier-Komplement: bilde erst das Einer-Komplement und addiere dann eine Eins (modulo 2^n):

$$\overline{x}^2 = \overline{x_{n-1}}\ \overline{x_{n-2}}\ldots\overline{x_0} + 1 \text{ (modulo } 2^n).$$

Komplementdarstellung (2)

Beispiel:

$$\begin{aligned} x &= 10110100 \\ \overline{x}^1 &= 01001011 \\ \overline{x}^2 &= 01001100 \end{aligned}$$

Beispiel:

$$\begin{aligned} x &= 0001101101011000 \\ \overline{x}^1 &= 1110010010100111 \\ \overline{x}^2 &= 1110010010101000 \end{aligned}$$

Komplementdarstellung (3)

Für jede b-adische Zahlendarstellung kann das $(b-1)$- und das b-Komplement definiert werden.

Eine Komplementdarstellung ist auf eine beliebige, aber **fest vorgegebene Stellenzahl** bezogen.

Komplementdarstellung: eine negative Zahl $-x$ wird durch die Differenz $N - x$ dargestellt (N = Anzahl darstellbarer Zahlen)

Beispiel: Für $b = 10$ und $n = 3$ gilt $N = 10^3 = 1000$. Im Zehner-Komplement: -23 entspricht $N - 23 = 977$

568	568
- 23	+ 977
545	1545

Komplementdarstellung (4)

Dualzahl	Einer-Komplement	Zweier-Komplement
000 = 0	000 = 0	000 = 0
001 = 1	001 = 1	001 = 1
010 = 2	010 = 2	010 = 2
011 = 3	011 = 3	011 = 3
100 = 4	100 = -3	100 = -4
101 = 5	101 = -2	101 = -3
110 = 6	110 = -1	110 = -2
111 = 7	111 = -0	111 = -1

Subtraktion im Einer-Komplement

Voraussetzung: x und y zwei n-stellige Dualzahlen

1. anstelle von $x - y$ berechne $x + \overline{y}^1$
2. ggf. Übertrag zur niederwertigsten Stelle addieren

Beispiel: Sei $n = 8$. Aus der Rechnung

x	01110111	$\widehat{=}$	119
$-y$	−00111011	$\widehat{=}$	59
	00111100	$\widehat{=}$	60

wird im Einer-Komplement

x	01110111		00111011
$+\overline{y}^1$	+11000100		+1
	100111011		00111100

Subtraktion im Einer-Komplement (2)

Beispiel: Aus der Rechnung

$$\begin{array}{rr} 27 & 00011011 \\ -38 & -00100110 \\ \hline -11 & 00001011 \end{array}$$

wird im Einer-Komplement

$$\begin{array}{rcr} 00011011 & \widehat{=} & 27 \\ +11011001 & \widehat{=} & -38 \\ \hline 11110100 & \widehat{=} & -11 \end{array}$$

Subtraktion im Zweier-Komplement

Voraussetzung: x und y zwei n-stellige Dualzahlen

1. anstelle von $x - y$ berechne $x + \overline{y}^2$
2. ignoriere eventuell auftretenden Übertrag

Beispiel: Sei $n = 8$. Aus der Rechnung

$$\begin{array}{rrcr} x & 01110111 & \widehat{=} & 119 \\ -y & -00111011 & \widehat{=} & 59 \\ \hline & 00111100 & \widehat{=} & 60 \end{array}$$

wird im Zweier-Komplement

$$\begin{array}{rr} x & 01110111 \\ +\overline{y}^2 & +11000101 \\ \hline & 100111100 \end{array}$$

Subtraktion im Zweier-Komplement (2)

Beispiel: Aus der Rechnung

$$\begin{array}{rr} 27 & 00011011 \\ -38 & -00100110 \\ \hline -11 & 00001011 \end{array}$$

wird im Zweier-Komplement

$$\begin{array}{rcr} 00011011 & \widehat{=} & 27 \\ +11011010 & \widehat{=} & -38 \\ \hline 11110101 & \widehat{=} & -11 \end{array}$$

Exzess-Darstellung

- zur Darstellung des Exponenten bei Gleitpunktzahlen
- zum Wert einer Zahl x wird eine positive Zahl q addiert, so dass das Ergebnis nicht negativ ist
- Exzess q gleich Betrag der größten negativen Zahl

Beispiel: Anzahl Bits gleich 4 $\Rightarrow q = 8$

x	$x+q$	Code	x	$x+q$	Code	x	$x+q$	Code
-8	0	0000	-3	5	0101	2	10	1010
-7	1	0001	-2	6	0110	3	11	1011
-6	2	0010	-1	7	0111	4	12	1100
-5	3	0011	0	8	1000	..	..	..
-4	4	0100	1	9	1001	7	15	1111

Darstellung von Gleitkommazahlen

Operationen auf Festkomma-Zahlen: Das Komma muss bei allen Operanden an der gleichen Stelle stehen.

Beispiel: Addition der Zahlen 101.01 und 11.101:

$$\begin{array}{rr} & 101\ .\ 01 \\ + & 11\ .\ 101 \\ \hline & 1000\ .\ 111 \end{array}$$

Also: Zahlen müssen eventuell transformiert werden
$\Rightarrow$ signifikante Stellen gehen verloren

Beispiel: Bei 4 Vor- und 4 Nachkommastellen muss die Zahl 0.000111101 durch 0000.0001 abgerundet dargestellt werden $\rightarrow$ fünf signifikante Stellen gehen verloren

Halblogarithmische Darstellung

Bei der **Gleitpunktdarstellung** wird jede Zahl z dargestellt in der Form:

$$z = \pm m \cdot b^{\pm d}$$

m : **Mantisse**
d : **Exponent**
b : **Basis des Exponenten**

Beispiele:

$$\begin{aligned} 3.14159 &= 0.314159 \cdot 10^{1} \\ 0.000021 &= 0.21 \cdot 10^{-4} \\ 12340000 &= 0.1234 \cdot 10^{8} \end{aligned}$$

Halblogarithmische Darstellung (2)

Die halblogarithmische Darstellung ist nicht eindeutig:

$$\begin{aligned} 3.14159 &= 0.0314159 \cdot 10^{2} \\ &= 0.314159 \cdot 10^{1} \\ &= 31.4159 \cdot 10^{-1} \\ &= 314.159 \cdot 10^{-2} \end{aligned}$$

Mehrdeutigkeiten vermeiden $\rightarrow$ normalisierte Darstellung

Eine Gleitkommazahl der Form $\pm m \cdot b^{\pm d}$ heißt **normalisiert**, wenn gilt:

$$\frac{1}{b} \leq |m| < 1$$

Halblogarithmische Darstellung (3)

Beispiele:

- $(0.000011101)_2 \rightarrow (0.11101)_2 \cdot 2^{-4}$
- $(1001.101)_2 \cdot 2^{10} \rightarrow (0.1001101)_2 \cdot 2^{14}$
- $3.14159 \rightarrow 0.314159 \cdot 10^{1}$
- $47.11 \cdot 10^{2} \rightarrow 0.4711 \cdot 10^{4}$
- $0.0815 \cdot 10^{-3} \rightarrow 0.815 \cdot 10^{-4}$

Gleitpunktzahlen im Rechner

Speichern einer Gleitkommazahl im Computer: aufteilen der 32 Bit wie folgt:

1. **Mantisse:** 23 Bit für den Betrag plus ein Bit für das Vorzeichen der Mantisse (Vorzeichen-/Betragdarstellung)
2. **Exponent:** 8 Bit (Zweier-Komplement Darstellung)
3. erste Stelle der Mantisse ist immer Null und wird in der Rechnerdarstellung ignoriert

analog für 64 Bit-Darstellung ...

Gleitpunktzahlen im Rechner (2)

Beispiel:

$$\begin{aligned}(5031.1875)_{10} &= (1001110100111.0011)_2 \cdot 2^0 \\ &= (0.10011101001110011)_2 \cdot 2^{13} \\ &= (0.10011101001110011)_2 \cdot 2^{(00001101)_2}\end{aligned}$$

Vorzeichen	:	0
Mantisse	:	10011101001110011000000
Exponent	:	00001101

Gleitpunktzahlen im Rechner (3)

Beispiel:

$$\begin{aligned}(-0.078125)_{10} &= (-0.000101)_2 \cdot 2^0 \\ &= (-0.101)_2 \cdot 2^{-3} \\ &= (-0.101)_2 \cdot 2^{(11111101)_2}\end{aligned}$$

Vorzeichen	:	1
Mantisse	:	10100000000000000000000
Exponent	:	11111101

Gleitpunktzahlen im Rechner (4)

Null im darstellbaren Bereich nicht enthalten $\Rightarrow$ abweichende Darstellung: Vorzeichen 0, Mantisse 0, Exponent 0

Ist die Basis b des Exponenten 2, so ist das erste Bit der Mantisse immer 1:

- erstes Bit der Mantisse nicht speichern (**hidden bit**)
- Verwechselung zwischen $\frac{1}{2}$ und 0 ausschließen

Gleitkommazahlen: Einbußen hinsichtlich Genauigkeit

- größte Gleitkommazahl bei 32 Bit: etwa 2^{127}
- größte Zahl in Dualdarstellung bei 32 Bit: $2^{31} - 1$

Gleitpunktzahlen: Arithmetik

Seien $x = m_x \cdot 2^{d_x}$ und $y = m_y \cdot 2^{d_y}$.

Multiplikation:
Mantissen multiplizieren, Exponenten addieren

$$x \cdot y = (m_x \cdot m_y) \cdot 2^{d_x + d_y}$$

Division:
Mantissen dividieren, Exponenten subtrahieren

$$x : y = (m_x : m_y) \cdot 2^{d_x - d_y}$$

Gleitpunktzahlen: Arithmetik (2)

Beispiel: $12.18 \cdot 3.7$

$$\begin{aligned} 0.1218 \cdot 10^2 \ \cdot \ 0.37 \cdot 10^1 &= 0.045066 \cdot 10^3 \\ &= 45.066 \end{aligned}$$

Beispiel: $45.066 : 3.7$

$$\begin{aligned} 0.45066 \cdot 10^2 \ : \ 0.37 \cdot 10^1 &= 1.218 \cdot 10^1 \\ &= 12.18 \end{aligned}$$

Gleitpunktzahlen: Arithmetik (3)

Seien $x = m_x \cdot 2^{d_x}$ und $y = m_y \cdot 2^{d_y}$.

Addition:

$$x + y = (m_x \cdot 2^{d_x - d_y} + m_y) \cdot 2^{d_y} \quad \text{falls} \quad d_x \leq d_y$$

Subtraktion:

$$x - y = (m_x \cdot 2^{d_x - d_y} - m_y) \cdot 2^{d_y} \quad \text{falls} \quad d_x \leq d_y$$

kleiner Exponent muss großem Exponenten angeglichen werden $\Rightarrow$ Rundungsfehler durch Denormalisierung

Rundungsfehler

Seien x, y, z wie folgt gegeben:

$$\begin{aligned} x &= +0.1235 \cdot 10^3 \\ y &= +0.5512 \cdot 10^5 \\ z &= -0.5511 \cdot 10^5 \end{aligned}$$

Dann gilt

$$\begin{aligned} x + y &= +0.1235 \cdot 10^3 + 0.5512 \cdot 10^5 \\ &= +0.0012 \cdot 10^5 + 0.5512 \cdot 10^5 \\ &= +0.5524 \cdot 10^5 \end{aligned}$$

$$\begin{aligned} (x + y) + z &= +0.5524 \cdot 10^5 - 0.5511 \cdot 10^5 \\ &= +0.0013 \cdot 10^5 \\ &= +0.1300 \cdot 10^3 \end{aligned}$$

Rundungsfehler (2)

Andererseits gilt

$$\begin{aligned} y+z &= +0.5512 \cdot 10^5 - 0.5511 \cdot 10^5 \\ &= +0.0001 \cdot 10^5 \\ &= +0.1000 \cdot 10^2 \end{aligned}$$

$$\begin{aligned} x+(y+z) &= +0.1235 \cdot 10^3 + 0.1000 \cdot 10^2 \\ &= +0.1235 \cdot 10^3 + 0.0100 \cdot 10^3 \\ &= +0.1335 \cdot 10^3 \\ &\neq +0.1300 \cdot 10^3 \end{aligned}$$

Ungenauigkeiten bei Gleitpunktzahlen

Ungenauigkeiten im Umgang mit Gleitpunktzahlen

- bei der Umwandlung vom Dezimal- ins Dualsystem
- und bei den arithmetischen Operationen.

In der Regel spielen kleine Abweichungen keine große Rolle. Im Rechner werden oft tausende von Rechenoperationen hintereinander ausgeführt: kleine Rundungsfehler addieren sich, Resultat wird völlig unbrauchbar!

IEEE-754: Gleitpunktzahlen

- Exponent in Exessdarstellung speichern
- normalisiert wird auf 1.xxxxxx
- die führende Eins wird nicht abgespeichert
- einfache Genauigkeit (32Bit)
  * Exzess: $2^7 - 1 = 127$
  * 1Bit Vorzeichen, 8 Bit Exponent, 23 Bit Mantisse
- doppelte Genauigkeit (64Bit)
  * Exzess: $2^{10} - 1 = 1023$
  * 1Bit Vorzeichen, 11 Bit Exponent, 52 Bit Mantisse

IEEE-754: Gleitpunktzahlen (2)

Beispiel: einfache Genauigkeit

$$(-0.078125)_{10} = (-0.000101)_2 \cdot 2^0 = (-1.01)_2 \cdot 2^{-4}$$

Vorzeichen	:	1
Exponent	:	01111011 (≙ -4 + 127)
Mantisse	:	01000000000000000000000

Ergänzende Literatur:
Walter Oberschelp und Gottfried Vossen: Rechneraufbau und Rechnerstrukturen, Oldenbourg Verlag

Die Programmiersprache C

Historie

Entwicklung der Programmiersprache C:

- ist eng mit der Entwicklung des Betriebssystems UNIX verbunden
- UNIX und die meisten Programme, die damit eingesetzt werden, sind in C geschrieben
- C ist nicht von einem bestimmten Betriebssystem oder einer bestimmten Maschine abhängig

wesentliche Entwicklungsarbeiten zu UNIX und C: Anfang der 70er Jahre von Ken Thompson und Dennis Ritchie an den Bell Laboratorien

C basiert auf den Sprachen BCPL und B (beide typenlos)

Erste Schritte

Datentypen in C:

- Zeichen: `char`
- ganze Zahlen: `int`, `short int`, `long int`
- Gleitpunktzahlen: `float`, `double`

abgeleitete Datentypen können erzeugt werden mit:

- Zeigern
- Vektoren (auch: Array oder Felder)
- Strukturen
- Vereinigungen

Zeiger erlauben maschinenunabhängige Adress-Arithmetik.

Erste Schritte (2)

Das folgende Programm enthält Beispiele für die meisten Grundelemente, aus denen die Sprache C aufgebaut ist.

„Hello, world!“ auf dem Bildschirm ausgeben:

```
/* Einfügen von Standard-Bibliotheken */
#include <stdio.h>
#include <stdlib.h>

/* Durch main wird das Hauptprogramm eingeleitet */
main() {
    char *str = "world";
    printf("Hello, %s!\n", str);
    exit(0);
}
```

Erste Schritte (3)

- Durch `/*` und `*/` werden Kommentare eingeschlossen.
- Mittels `#include <stdio.h>` wird eine Bibliothek bereitgestellt, die Funktionen zur Ein- und Ausgabe enthält.
- Der Start eines Programms besteht im Ausführen der Funktion `main`.
- `char *str = "world"` definiert eine Variable `str`. Alle in einem C-Programm benutzten Variablen müssen explizit deklariert werden, wobei der Typ und der Name der Variablen festgelegt werden.
- Die Funktion `printf()` gibt eine Zeichenkette auf dem Bildschirm aus. Solche Standardfunktionen sind übersetzte Funktionen, die zur C-Implementierung gehören.

Erste Schritte (4)

- Mit `exit(0)` wird das Programm verlassen und der Wert 0 an den Kommandointerpreter zurückgegeben.
- Alle Anweisungen werden mit einem Semikolon beendet.
- Anweisungsfolgen werden mit geschweiften Klammern zusammengefasst, der geklammerte Block gilt als eine Anweisung.

Kriterien bei der Programmentwicklung

- Korrektheit bzgl. der Aufgabenstellung
- Handhabung, Benutzerführung, Online-Hilfe
- Technische Qualität
  * Laufzeit
  * Verbrauch von Ressourcen (Speicherplatz)
- Wirtschaftliche Qualität
  * Wartbarkeit
  * Erweiterbarkeit
  * Entwicklungskosten

Software-Qualität in ISO 9126 beschrieben

Programmieren

- **Editieren:**
  * C-Programm mit beliebigem Text-Editor eingeben
  * Programm kann aus mehreren Quelltext-Dateien bestehen (siehe „Modulare Programmierung“)
- **Übersetzen:**
  * Bei jeder Übersetzung wird zunächst automatisch ein **Präprozessor** aufgerufen, der eine Vorverarbeitung des Textes vornimmt → `#include <stdio.h>` ersetzen
  * Enthält der Quelltext keine Syntaxfehler, erzeugt der C-Compiler eine **Objektdatei**, auch Modul genannt.
  * Objektdatei: enthält Maschinencode und zusätzliche Informationen für das Binden.

Programmieren (2)

- **Binden:**
  * Die vom Compiler erzeugten Objektdateien werden durch den **Linker** zu einem lauffähigen Programm gebunden.
  * Beispiel: Verwendete Standardbibliotheken und das Programm werden zu einem ausführbaren Programm zusammengefügt.
- **Ausführen:** Das übersetzte und vollständig gebundene Programm kann ausgeführt und getestet werden.

Wie diese Schritte auszuführen sind, hängt sowohl vom Betriebssystem als auch von der C-Implementierung ab.

Vom Programm zur Maschine

Programme, in einer höheren Programmiersprache, müssen in eine Folge von Maschinenbefehlen übersetzt werden.

Die Übersetzung eines Programmtextes in eine Folge von Maschinenbefehlen wird vom **Compiler** durchgeführt.

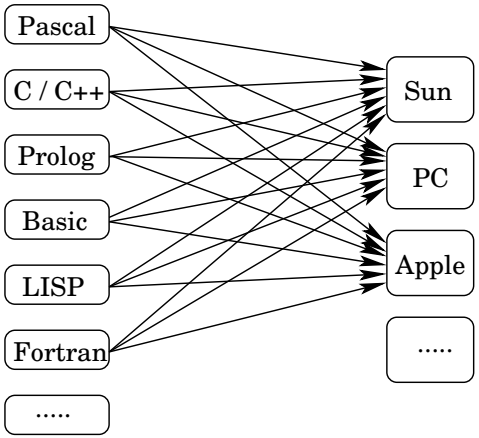
Das Ergebnis ist ein **Maschinenprogramm**, das in einer als **ausführbar** (engl. **executable**) gekennzeichneten Datei gespeichert ist.

Eine ausführbare Datei muss von einem **Ladeprogramm** in den Speicher geladen werden, um ausgeführt zu werden.

Ladeprogramme sind Teil des Betriebssystems, der Benutzer weiß in der Regel gar nichts von deren Existenz.

Vom Programm zur Maschine (2)

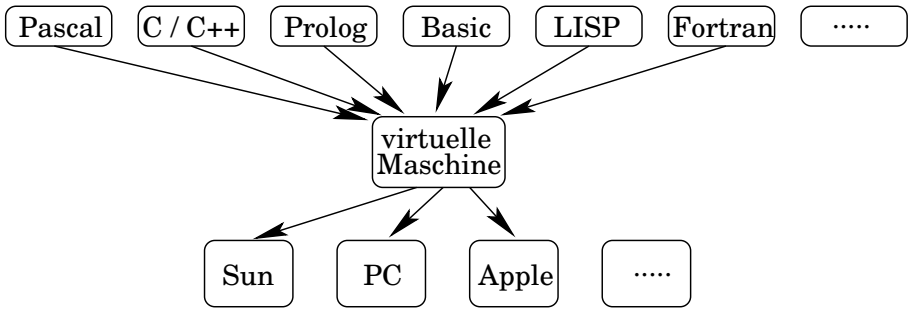
n Sprachen und m Rechnertypen $\Rightarrow$ $n \cdot m$ Compiler



Virtuelle Maschinen

Es würden $n + m$ Compiler ausreichen, wenn

- Code für eine virtuelle Maschine erzeugt wird, und
- die virtuelle Maschine auf allen konkreten Maschinen emuliert (in Software nachgebildet) wird.



Virtuelle Maschinen (2)

Probleme:

- Einige Sprachen oder Maschinentypen könnten bevorzugt werden.
- Geschwindigkeit bei der Programmausführung wird beeinträchtigt.

Für Java und C# sind virtuelle Maschinen auf verschiedenen Plattformen realisiert.

Compiler-Entwickler arbeiten mit **Zwischensprachen**, um unabhängig von der Quellsprache zu sein. Problem: Die Zwischensprache muss hinreichend allgemein sein (näheres dazu in der Vorlesung *Compilerbau* bei Prof. Dr. Becker).

Programmieren und Testen

Fehler, die während des Übersetzungsvorganges erkannt werden:

- **Syntaxfehler:** ist Rechtschreib- oder Grammatikfehlern vergleichbar (vertippt, falscher Satzbau)
- **Typfehler:** wenn nicht zueinander passende Dinge verknüpft werden (addieren des Straßennamens auf die Hausnummer)

Wurde das Programm fehlerlos übersetzt, kann es ausgeführt und getestet werden.

Programmieren und Testen (2)

Fehler, die nicht durch den Compiler erkannt werden:

- **Laufzeitfehler**
  - zulässige Wertebereiche werden überschritten
  - es wird durch 0 dividiert
  - es wird die Wurzel aus einer negativen Zahl gezogen
- **Denkfehler** werden sichtbar, wenn das Programm nicht das tut, was es tun soll.

Testen zeigt nur die Anwesenheit von Fehlern, nicht deren Abwesenheit!

Funktionalität hat höchste Priorität
⇒ **Tests sind immer durchzuführen**

Vom Problem zum Algorithmus

Ein Algorithmus ist eine präzise Vorschrift, um

- aus vorgegebenen Eingaben
- in endlich vielen Schritten
- eine bestimmte Ausgabe zu ermitteln.

(Abu Jáfar Hohammed ibu Musa al-Chowarizmi)

Programmieren ist das Umsetzen eines Algorithmus in eine für den Computer verständliche und ausführbare Form.

Problem ⟶ Algorithmus ⟶ Programm

Erst denken, dann programmieren!

Top-down Entwurf
Sortieren durch Einfügen (Insertion Sort)

Annahme:

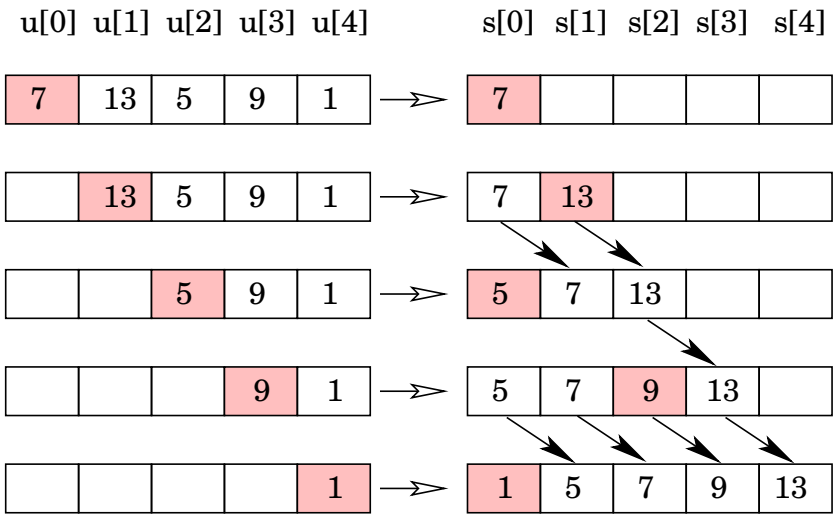
- Objekte sind im Array $u[0] \ldots u[n-1]$ gespeichert.
- Schlüssel stehen in $u[i].key$, Informationen in $u[i].data$.
- sortierte Objekte stehen hinterher in $s[0] \ldots s[n-1]$.

Algorithmus:

for $i := 0$ to $n-1$ step 1 do
    füge $u[i]$ am richtigen Platz in $s[0] \ldots s[i]$ ein

Top-down Entwurf (2)

Insertion Sort: Prinzip



Top-down Entwurf (3)

Das Einfügen von $u[i]$ in die Folge $s[0] \ldots s[i]$ muss genauer spezifiziert werden → **schrittweise verfeinern**

- suche die Stelle p, an der eingefügt werden soll
- verschiebe Zahlen ab Position p eine Stelle nach rechts
- füge $u[i]$ an der Position p ins Array ein

Top-down Entwurf (4)

for $i := 0$ to $n-1$ step 1 do

    ** suche die Stelle p, an der eingefügt werden soll **
    $p := 0$
    while $(u[i].key > s[p].key)$ and $(p < i)$ do
        $p := p + 1$

    ** verschiebe Zahlen ab Position p nach rechts **
    for $j := i - 1$ down to p step 1 do
        $s[j+1] := s[j]$

    ** füge Objekt an Position p ins Array ein **
    $s[p] := u[i]$

Top-down Entwurf (5)

Anmerkungen:

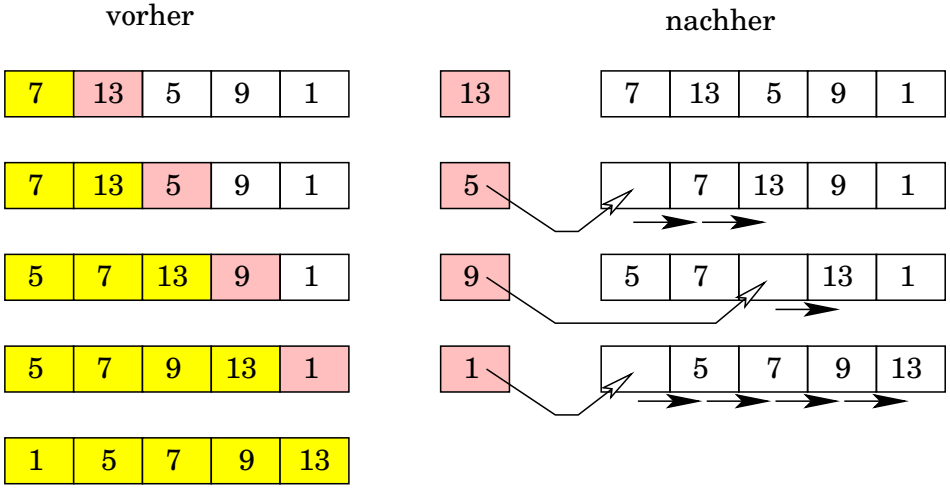
- Elemente in u und $s \rightarrow$ Speicherplatzverschwendung
- Verschieben der Zahlen kostet viel Zeit
- zusammenfassen: Position suchen/Zahlen verschieben

Insertion Sort Variante: am Ort sortieren

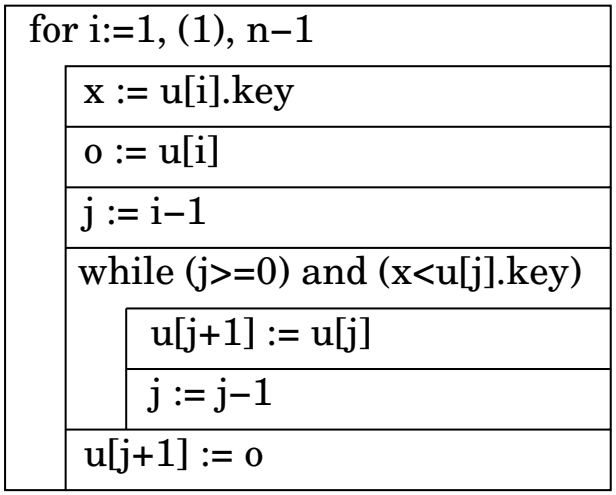
for $i := 1$ to $n-1$ step 1 do
  $x := u[i].key$
  $o := u[i]$
  $j := i-1$
  while $(j \geq 0)$ and $(x < u[j].key)$ do
    $u[j+1] := u[j]$
    $j := j-1$
  $u[j+1] := o$

Top-down Entwurf (6)

Insertion Sort Variante: Prinzip



Struktogramm: Insertion Sort



Sortieren: Hauptprogramm

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

int liste[50];

main() {
    int numberOfElements;

    numberOfElements = getData();
    insertionSort(numberOfElements);
    dataToScreen(numberOfElements);

    exit(0);
}
```

Sortieren: Daten einlesen

```
int getData() {
    int i, n;

    printf("Insertion Sort\n");
    printf("Wieviele Zahlen? (maximal 50)\n");
    scanf("%d", &n);

    for (i = 0; i < n; i++) {
        printf("%2d. Zahl eingeben: ", i + 1);
        scanf("%d", &liste[i]);
    }
    return n;
}
```


Sortieren: Sortierprozedur

```
void insertionSort(int n) {
    int i, j, x;
    for (i = 1; i < n; i++) {
        x = liste[i];
        j = i - 1;
        while ((j >= 0) && (x < liste[j])) {
            liste[j + 1] = liste[j];
            j = j - 1;
        }
        liste[j + 1] = x;
    }
    return;
}
```


Sortieren: Daten ausgeben

```
void dataToScreen(int n) {
    int i;

    printf("\nSortierte Zahlen\n");
    for (i = 0; i < n; i++) {
        printf("%2d. Zahl: %d\n", i + 1, liste[i]);
    }
}
```


Top-down Entwurf
Berechnung von Pi

$$\frac{\text{Viertelkreisfläche}}{\text{Quadratfläche}} = \frac{\text{Treffer im Viertelkreis}}{\text{Treffer im Quadrat}}$$

mit $r = 1$ und $A_{Kreis} = r^2 \cdot \pi$ ergibt sich:

$$\pi = 4 \cdot \frac{\text{Treffer im Viertelkreis}}{\text{Treffer im Quadrat}}$$

Algorithmus

treffer := 0
wiederhole n mal
    erzeuge Zufallszahlen x, y zwischen 0 und 1
    falls Punkt (x,y) innerhalb vom Viertelkreis
    dann treffer := treffer + 1
pi = 4 * treffer / n

Top-down Entwurf (2)

schrittweise verfeinern

- wiederhole n mal
  `for (i = 0; i < n; i++)`
- erzeuge Zufallszahlen x, y zwischen 0 und 1

```
x = 1.0 * rand() / RAND_MAX;
y = 1.0 * rand() / RAND_MAX;
```

- falls Punkt (x,y) innerhalb vom Viertelkreis
  `if (sqrt(x*x + y*y) < 1)`

Top-down Entwurf (3)

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

int main(void) {
    int i;
    int n = 100000;
    int treffer = 0;
    double x, y;
    double pi;

    srand(0);
```


Top-down Entwurf (4)

```
    for (i = 0; i < n; i++) {
        x = 1.0 * rand() / RAND_MAX;
        y = 1.0 * rand() / RAND_MAX;
        if (sqrt(x*x + y*y) <= 1)
            treffer += 1;
    }
    pi = 4.0 * treffer / n;
    printf("pi = %8.6f\n", pi);

    return 0;
}
```


Variablen

Um sinnvolle C-Programme schreiben zu können, ist es notwendig,

- Zwischenwerte zu speichern und
- diese Werte in weiteren Berechnungen zu verwenden.

Für die Speicherung der Werte steht der Hauptspeicher zur Verfügung. Ohne höhere Programmiersprachen:

- An welcher Stelle im Speicher steht der Wert?
- Wie viele Bytes gehören zu dem Wert?
- Welche Speicherplätze sind noch frei? ...

Variablen sind Behälter für Werte eines bestimmten Datentyps. Der Compiler setzt jeden Bezug auf eine Variable in die entsprechende Hauptspeicheradresse um.

Grundelemente der Sprache

Variablen und **Konstanten**: grundsätzliche Datenobjekte, die ein Programm manipuliert

Vereinbarungen:

- führen Variablen ein, die benutzt werden dürfen
- legen den Typ der Variablen fest und ggf. den Anfangswert

Operatoren kontrollieren, was mit den Werten geschieht.

In **Ausdrücken** werden Variablen und Konstanten mit Operatoren verknüpft, um neue Werte zu produzieren.

Der **Datentyp** eines Objekts legt seine Wertemenge und die Operatoren fest, die darauf anwendbar sind.

Zeichensatz

Folgende Zeichen sind in C-Programmen zulässig:

- Alphanumerische Zeichen:

```
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
a b c d e f g h i j k l m n o p q r s t u v w x y z
0 1 2 3 4 5 6 7 8 9
```

- Leerzeichen und Zeilenendezeichen
- Sonderzeichen:

```
( ) [ ] { } < >  + - * / % ^ ~ & | _
= ! ? # \ , . ; : ’ "
```


Zeichensatz (2)

- Steuerzeichen:
  - `\a` Gong-Zeichen (Bell)
  - `\b` ein Zeichen zurück (Backspace)
  - `\t` horizontaler Tabulator (Tab)
  - `\f` Seitenvorschub (Formfeed)
  - `\r` Wagenrücklauf (Carriage Return)
  - `\n` Zeilentrenner (New Line)
  - `\"` doppelte Anführungsstriche
  - `\’` einfacher Anführungsstrich
  - `\\` Backslash

Schlüsselwörter

Die folgenden Wörter haben eine vordefinierte Bedeutung:

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

Bezeichner und Namen

Bezeichner dienen zur **eindeutigen Identifizierung** von Objekten innerhalb eines Programms.

Ein Objekt in C ist ein Speicherbereich, der aus einer zusammenhängenden Folge von einem oder mehreren Bytes bestehen muss. (In OOP ist der Begriff eines Objekts anders definiert!)

Funktionsnamen, Variablennamen und andere Namen (z.B. Sprungmarken) sind Folgen von Zeichen, bestehend aus

- Buchstaben,
- Ziffern und
- dem Unterstrich.

Groß-/Kleinschreibung wird unterschieden.

Bezeichner und Namen (2)

Eigenschaften:

- Ein Bezeichner beginnt immer mit einem Buchstaben oder einem Unterstrich. Namen vermeiden, die mit zwei Unterstrichen beginnen → evtl. systemintern benutzt!
- Bezeichner dürfen nicht mit Schlüsselwörtern wie `int`, `while` oder `if` übereinstimmen.
- Die Anzahl der Zeichen in Bezeichnern ist in einigen Implementierungen begrenzt. Der Standard definiert für
  * interne Namen: die ersten 31 Zeichen sind signifikant
  * externe Namen: mind. 6 Zeichen werden unterschieden, jedoch nicht notwendig Groß-/Kleinschreibung

Bezeichner und Namen (3)

Anmerkung: Benennen Sie Variablen so,

- dass der Name den Zweck der Variablen andeutet, und
- eine Verwechselung durch Tippfehler unwahrscheinlich ist.

Datentypen

C ist eine **typisierte** Programmiersprache: Alle in einem C-Programm verwendeten Größen (Konstanten, Variablen, Funktionen) haben einen Typ (im Gegensatz zu Skript-Sprachen wie Perl).

Datentypen (2)

Der Typ von Konstanten (`#define ...`) ergibt sich in der Regel aus deren Wert.

Die Typen von Variablen und Funktionen werden in deren Deklaration bzw. Definition festgelegt.

Vorteil: Da der Compiler Unverträglichkeiten von Typen erkennen kann, werden durch die Typisierung Fehlerquellen verringert.

Die Größen der primitiven Datentypen sind nicht festgelegt, sie sind abhängig von der C-Implementierung und von der Wortlänge des Rechners
→ führt zu Problemen bei Portierungen, anders in Java

Elementare Datentypen

Der **Integer-Typ** `int` stellt ganzzahlige Werte mit/ohne Vorzeichen dar, die üblichen arithmetischen Operationen sind definiert.

- Typen: `short`, `int` und `long`.
- Versionen: `signed` und `unsigned`.
- Größen: `short` $\geq$ 16 Bit, `int` $\geq$ 16 Bit, `long` $\geq$ 32 Bit.

Der **Character-Typ** `char` wird zur Darstellung von einzelnen Zeichen bzw. Zeichenketten (Strings) verwendet.

Der **Fließkommatyp** `float` stellt reelle Zahlen dar.

- `float`: einfach-genaue Fließkommawerte
- `double`: mindestens einfach-genaue Fließkommawerte
- `long double`: mindestens so groß wie `double`

Elementare Datentypen (2)

Der **leere Typ** `void` ist ein besonderer Typ, der keinen Wert hat und auf dem keine Operationen definiert sind. Er steht für die leere Menge.

Bei Funktionen wird mittels `void` definiert, dass kein Rückgabewert geliefert wird (Prozedur) bzw. keine Parameter übergeben werden.

Durch Gruppierungen wie Felder (array), Verbund (union) und Strukturierung (struct) können neue Typen gebildet werden.

Character-Typ

Der Datentyp `char` hat die Größe ein Byte und kann ein Zeichen aus dem Zeichensatz der Maschine speichern.

Der Wert einer Zeichenkonstanten ist der numerische Wert des Zeichens im Zeichensatz der Maschine (bspw. ASCII).

Zeichen sind ganzzahlig, arithmetische Operationen sind definiert. **Beispiel:** `'a' + 'b' = Ã` ($\rightarrow 97 + 98 = 195$)

Schreibweise:

- ein Zeichen innerhalb von einfachen Anführungszeichen
- Ersatzdarstellungen für Steuerzeichen (`\n`, `\t`, ...)
- oktale Darstellung `'\ooo'`: ein bis drei oktale Ziffern
- hexadezimal `'\xnn'`: ein oder zwei Hex-Ziffern

Character-Typ (2)

Beispiele:

```
/* 'X' in ASCII  */
char c1 = 'X', c2 = '\130', c3 = '\x58';
/* Sonderzeichen */
char c4 = '\"', c5 = '\\', c6 = '\'';
/* Steuerzeichen */
char c7 = '\n', c8 = '\t', c9 = '\r';
```

konstante Zeichenkette: Eine Folge von beliebig vielen Zeichen, die von doppelten Anführungszeichen umgeben ist.

```
"Eine konstante Zeichenkette"
```

Character-Typ (3)

Konstante Zeichenketten können aneinandergehängt werden, um sie im Programm auf mehrere Zeilen zu verteilen:

```
"Eine " "konstante" " Zeichenkette"
```

Eine Zeichenkette ist ein Vektor (oder Array) von Zeichen. Intern hat jede Zeichenkette am Ende ein Null-Zeichen (`'\0'`), wodurch die Länge prinzipiell nicht begrenzt ist.

Integer-Typ

Schreibweise:

- Buchstabe l oder L am Ende bedeutet `long`
- Buchstabe u oder U am Ende bedeutet `unsigned`
- Ziffer 0 am Anfang bedeutet oktal
- Zeichen 0x oder 0X am Anfang bedeuten hexadezimal

Beispiele:

- `123456789UL`
- `022` bedeutet $(22)_8 = (18)_{10}$
- `0x1F` bedeutet $(1F)_{16} = (31)_{10}$

Fließkomma-Typ

Schreibweise:

- Suffix f oder F vereinbart `float`
- Suffix l oder L vereinbart `long double`
- Dezimalpunkt und/oder Exponent vorhanden

Beispiele:

- 1e-3 $= 1 \cdot 10^{-3} = 0,001$
- 234E4 $= 234 \cdot 10^4 = 2.340.000$
- -12.34e-67F
- 234.456
- .123e-42

Vereinbarungen

Alle Variablen müssen vor Gebrauch vereinbart (deklariert) werden. Eine Vereinbarung gibt einen Typ an und enthält eine Liste von einer oder mehreren Variablen dieses Typs:

```
int lower, upper, step;
char c, line[256];
```

Variablen können beliebig auf mehrere Vereinbarungen verteilt werden → Vereinbarung kann kommentiert werden

```
int lower;
int upper;
int step;
char c;
char line[256];
```

Wenn eine Vereinbarung kommentiert werden muss, dann ist der Name der Variablen schlecht gewählt!

Vereinbarungen (2)

Eine Variable kann bei ihrer Vereinbarung auch initialisiert werden.

Beispiel: `double epsilon = 1.0e-5;`

Mit dem Attribut `const` kann bei der Vereinbarung einer Variablen angegeben werden, dass sich ihr Wert nicht ändert.

Beispiel: `const double e = 2.71828182845904523536;`

Bei einem Vektor bedeutet `const`, dass die Elemente nicht verändert werden.

Beispiel: `const int arr[] = {1, 2, 3, 4};`
Dann ist zwar eine Anweisung wie `arr[0] = 0;` verboten, aber Elemente durch `int *z = arr;` über Zeiger `z` änderbar

Vereinbarungen (3)

Ändern einer `const`-Variablen: Resultat implementierungsabhängig!

- gcc → Warning: assignment of read-only variable
  aber: Laufzeitfehler!
- Borland C++→ Error: Cannot modify a const object.

Es ist möglich, einen Zeiger als `const` zu definieren → keine Adressarithmetik möglich, aber der Inhalt kann geändert werden:

```
int i = 10;
int* const zi = &i;

*zi += 1;   /* i = 11   */
zi += 1;    /* verboten */
```


Arithmetische Operatoren

Operator	Beispiel	Bedeutung
`+`	`+i`	positives Vorzeichen
`-`	`-i`	negatives Vorzeichen
`+`	`i+5`	Addition
`-`	`i-j`	Subtraktion
`*`	`i*8`	Multiplikation
`/`	`i/5`	Division
`%`	`i%6`	Modulo
`=`	`i = 5+j`	Zuweisung
`+=`	`i += 5`	`i = i+5`
`-=`	`i -= 6`	`i = i-6`
`*=`	`i *= 5`	`i = i*5`
`/=`	`i /= 7`	`i = i/7`

Arithmetische Operatoren (2)

Anmerkungen:

- der Operator `%` kann **nicht** auf `float`- oder `double`-Werte angewendet werden.
- negative Operanden → maschinenabhängiges Verhalten
  * In welcher Richtung wird bei `/` abgeschnitten?
    Beispiel: `-15 / 2 = -7` oder `-15 / 2 = -8` ???
  * Welches Vorzeichen hat das Resultat von `%`?
    Beispiel: `-15 % 12 = -3` oder `-15 % 12 = 9` ???

  → Probleme bei Portierungen!

Arithmetische Operatoren (3)

Anmerkungen: (Fortsetzung)

- Vorrang der Operatoren in abnehmender Reihenfolge:
  1. unäre Operatoren + und - (Vorzeichen)
  2. *, / und %
  3. binäre Operatoren + und -

  Beispiel: `5 * -7 - 3` → `(5 * (-7)) - 3`
- Arithmetische Operationen werden von links her zusammengefasst.

  Beispiel: `1 + 3 + 5 + 7` → `((1 + 3) + 5) + 7`

  zur Erinnerung: die Addition auf Gleitkommazahlen ist nicht assoziativ aufgrund von Rundungsfehlern bei der Denormalisierung

Inkrement- und Dekrementoperatoren

Ausdruck	Bedeutung
`++i`	`i` wird um den Wert 1 erhöht, **bevor** `i` im Ausdruck weiterverwendet wird (Präfix-Notation)
`--i`	`i` wird um den Wert 1 vermindert, **bevor** `i` im Ausdruck weiterverwendet wird (Präfix-Notation)
`i++`	`i` wird um den Wert 1 erhöht, **nachdem** `i` im Ausdruck weiterverwendet wird (Postfix-Notation)
`i--`	`i` wird um den Wert 1 vermindert, **nachdem** `i` im Ausdruck weiterverwendet wird (Postfix-Notation)

Inkrement- und Dekrementoperatoren (2)

Diese Operatoren können nur auf Variablen angewendet werden, nicht auf Ausdrücke. **Verboten:** `(i+j)++`

Ausdrücke werden unter Umständen schwer einsichtig:

```
int x, y;   /* Variablendeklaration */

x = 1;
y = ++x + 1;
            /* hier: x = 2, y = 3 */
x = 1;
y = x++ + 1;
            /* hier: x = 2, y = 2 */
x = 1;
x = ++x + 1;
            /* hier: x = 3 */
```


Seiteneffekte

Der Wert eines Ausdrucks ist oft von Variableninhalten abhängig. Die Auswertung des Ausdrucks ändert aber nicht den Inhalt der Variablen:

- `(x+1) * (x+1)` kann durch `sqr(x+1)` ersetzt werden
- `(x+1) - (x+1)` kann durch 0 ersetzt werden

Schreibt man `++x` anstelle von `x+1`, so ergeben sich unter Umständen seltsame Gleichungen: `(++x) - (++x) = -1`

`++x` hat hier den Seiteneffekt, dass der Inhalt von x um 1 erhöht wird.

Anmerkung: Die Auswertung eines Ausdrucks soll einen Wert liefern, aber keinen Seiteneffekt haben.

Vergleichsoperatoren

Op	Beispiel	Bedeutung
<	`i < 7`	kleiner als
<=	`i <= 7`	kleiner gleich
==	`i == 7`	gleich

Op	Beispiel	Bedeutung
>	`i > j`	größer als
>=	`i >= j`	größer gleich
!=	`i != j`	ungleich

Prioritäten:

- Vergleichsoperatoren <, >, <=, >= haben gleiche Priorität
- Äquivalenzoperatoren ==, != haben geringere Priorität
- Vergleiche: geringerer Vorrang als arithm. Operatoren

Beispiele:

- `i < l - 1` wird bewertet wie `i < (l - 1)`
- `2+2 < 3 != 5 > 7` entspricht `(4 < 3) != (5 > 7)`

Boolesche Werte in C

In C wird

- **false** durch den Wert 0 und
- **true** durch einen Wert ungleich 0 dargestellt.

$\Rightarrow$ **bizarre Ausdrücke möglich:** `3 < 2 < 1` ist **true**, denn `(3 < 2)` = **false** = 0 und `0 < 1`.

Oft findet man in C-Programmen verkürzte Anweisungen.

Beispiel:

- `while (x)` $\Longleftrightarrow$ `while (x != 0)`
- `while (strlen(s))` $\Longleftrightarrow$ `while (strlen(s) > 0)`

Logische Verknüpfungen

Op	Beisp.	Ergebnis (Bedeutung)
&&	`a && b`	a und b wahr, dann 1, sonst 0 (log. UND)
||	`a || b`	a oder b wahr, dann 1, sonst 0 (log. ODER)
!	`!a`	liefert 1, falls a falsch ist, sonst 0

Prioritäten:

- && hat Vorrang vor ||
- Vergleichs- und Äquivalenzoperatoren: höherer Vorrang

Beispiele:

- `i < l-1 && c != EOF` $\rightarrow$ keine Klammern notwendig
- `!valid` $\Longleftrightarrow$ `valid == 0`

Verkürzte Auswertung

Ausdrücke werden nur solange bewertet, bis das Ergebnis feststeht!

Es gilt:

- `X && Y == 0`, falls `X == 0`
- `X || Y == 1`, falls `X == 1`

Verkürzte Auswertung ist notwendig, um Laufzeitfehler zu vermeiden.

Beispiele:

- `(x != 1) && (1/(x-1) > 0)`
- `scanf("%d", &n) > 0 || exit(errno)`

Sonstige Operatoren in C

was es nicht gibt:

- keine Operationen, mit denen zusammengesetzte Objekte wie Zeichenketten, Mengen, Listen oder Vektoren direkt bearbeitet werden können
- keine Operationen, die einen ganzen Vektor oder eine Zeichenkette manipulieren, jedoch Struktur als Einheit kopierbar
- keine Ein- und Ausgabe, keine eingebauten Techniken für Dateizugriff

$\Rightarrow$ abstrakte Mechanismen müssen als explizit aufgerufene Funktionen zur Verfügung gestellt werden

C-Implementierungen enthalten eine relativ standardisierte Sammlung solcher Funktionen (ANSI-Standard).

Typumwandlungen

Operator mit Operanden unterschiedlichen Typs: Werte werden in gemeinsamen Datentyp umgewandelt!

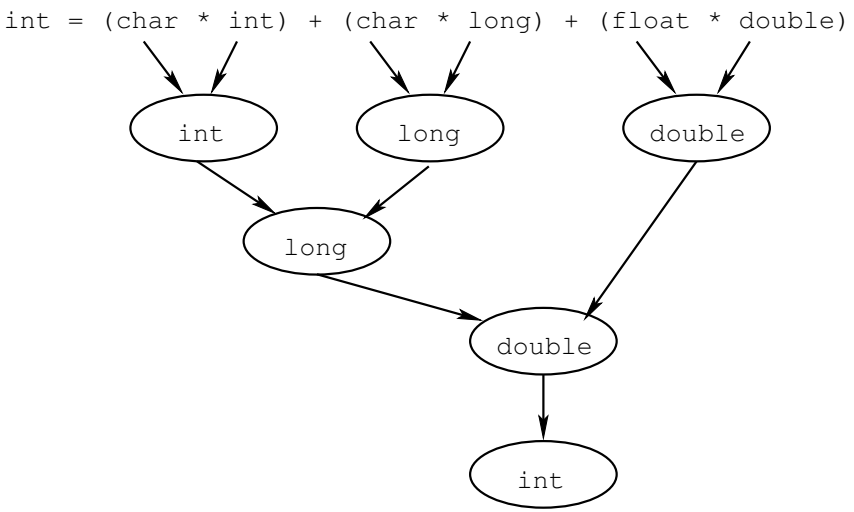
Implizite Typumwandlungen nur, wenn „kleiner“ Operand in „großen“ umgewandelt wird, ohne dabei Informationen zu verlieren (z.B. `int` nach `double`).

Ausdrücke, die zu Informationsverlust führen könnten, sind nicht verboten, können aber eine Warnung hervorrufen:

- Zuweisung eines langen Integer-Typs an einen kurzen
- Zuweisung eines Gleitpunkt-Typs an einen Integer-Typ

Sinnlose Ausdrücke, wie ein `float`-Wert als Vektorindex, sind verboten.

implizite Typumwandlungen



explizite Typumwandlungen

Typumwandlung mit unärer Umwandlungsoperation (**cast**) explizit erzwingen:

```
int a = 1, b = 2;
float x;

x = a / b;          /* x == 0   */
x = (float)a / b;   /* x == 0.5 */
```

Operatoren zur Bitmanipulation

Ganze Zahlen können als Bitvektoren aufgefasst werden:

```
    ...
-3  =  1111 1111 1111 1101
-2  =  1111 1111 1111 1110
-1  =  1111 1111 1111 1111
 0  =  0000 0000 0000 0000
 1  =  0000 0000 0000 0001
 2  =  0000 0000 0000 0010
 3  =  0000 0000 0000 0011
    ...
```

Manipulation einzelner Bits in C: Shift-Operatoren >> und << sowie logische Operatoren `&`, `|`, `^` und `~`

Operatoren zur Bitmanipulation (2)

nur auf Integer-Typen anwendbar:

Op	Beispiel	Bedeutung
<<	`i << j`	Links-Shift von `i` um `j` Stellen
>>	`i >> j`	Rechts-Shift von `i` um `j` Stellen
&	`i & j`	Bitweises UND von `i` und `j`
|	`i | j`	Bitweises ODER von `i` und `j`
^	`i ^ j`	Bitweises Exklusiv-ODER von `i` und `j`
~	`~i`	Einerkomplement von `i`

Prioritäten:

- Shift-Operatoren vor Äquivalenzoperatoren `==` und `!=`
- Äquivalenzoperatoren Vorrang vor `&`, `|` und `^`
- **Beispiel:** `(x & MASK) == 0` statt `x & MASK == 0`

Operatoren zur Bitmanipulation (3)

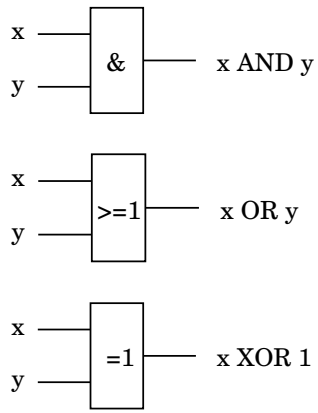
Beispiele:

```
short i = 1;          /* i = 0000 0000 0000 0001 */
i = i << 3;           /* i = 0000 0000 0000 1000 */
i = i >> 2;           /* i = 0000 0000 0000 0010 */
i = i | 5;            /* i = 0000 0000 0000 0111 */
i = i & 3;            /* i = 0000 0000 0000 0011 */
i = i ^ 5;            /* i = 0000 0000 0000 0110 */
i = ~i;               /* i = 1111 1111 1111 1001 */
```


Operatoren zur Bitmanipulation (4)

Beispiel: $x = 39 = (100111)_2$ und $y = 45 = (101101)_2$

```
  x   100111
  y = 101101
x&y = 100101

  x   100111
  y = 101101
x|y = 101111

  x   100111
  y = 101101
x^y = 001010
```



Shift-Operatoren

- schiebe einen Wert vom Typ `unsigned` nach rechts
  → es wird immer Null nachgeschoben

- schiebe vorzeichenbehafteten Wert nach rechts
  * **arithmetic shift:** Vorzeichenbit wird nachgeschoben
  * **logical shift:** Null-Bits werden nachgeschoben

Shift-Operatoren (2)

Mathematische Deutung für positive, ganze Zahlen:

- Links-Shift um m Stellen: Multiplikation mit 2^m
- Rechts-Shift, m Stellen: ganzzahlige Division durch 2^m

$z = (x_n x_{n-1} \ldots x_0)_2 \;\Rightarrow\; z = x_n \cdot 2^n + x_{n-1} \cdot 2^{n-1} + \cdots + x_0 \cdot 2^0$

Links-Shift oder Multiplikation mit 2:

$$\begin{aligned} z \cdot 2 &= x_n \cdot 2^{n+1} + x_{n-1} \cdot 2^n + \cdots + x_0 \cdot 2^1 + 0 \cdot 2^0 \\ &= (x_n\ x_{n-1}\ x_{n-2}\ \ldots\ x_0\ 0)_2 \end{aligned}$$

Rechts-Shift oder ganzzahlige Division durch 2:

$$\begin{aligned} z/2 &= x_n \cdot 2^{n-1} + x_{n-1} \cdot 2^{n-2} + \cdots + x_1 \cdot 2^0 \\ &= (x_n\ x_{n-1}\ x_{n-2}\ \ldots\ x_1)_2 \end{aligned}$$

Shift-Operatoren (3)

Beispiel: $17 = (00010001)_2$

$17 \cdot 2 = 34 = (00100010)_2$ $\quad \lfloor 17/2 \rfloor = 8 = (00001000)_2$
$17 \cdot 4 = 68 = (01000100)_2$ $\quad \lfloor 17/4 \rfloor = 4 = (00000100)_2$

Einerkomplement:

$$\begin{aligned} -17 &= (11101110)_2 \\ -17 \cdot 2 = -34 &= (11011101)_2 \rightarrow \textbf{kein Links-Shift!} \\ \lfloor -17/2 \rfloor = -8 &= (11110111)_2 \end{aligned}$$

Zweierkomplement:

$$\begin{aligned} -17 &= (11101111)_2 \\ -17 \cdot 2 = -34 &= (11011110)_2 \\ \lfloor -17/2 \rfloor = -8 &= (11111000)_2 \rightarrow \textbf{kein Rechts-Shift!} \end{aligned}$$

Shift-Operatoren (4)

Beispiel: $29 = (00011101)_2$

$29 \cdot 2 = 58 = (00111010)_2$ $\quad \lfloor 29/2 \rfloor = 14 = (00001110)_2$
$29 \cdot 4 = 116 = (01110100)_2$ $\quad \lfloor 29/4 \rfloor = 7 = (00000111)_2$

Einerkomplement:

$$\begin{aligned} -29 &= (11100010)_2 \\ -29 \cdot 2 = -58 &= (11000101)_2 \rightarrow \textbf{kein Links-Shift!} \\ \lfloor -29/2 \rfloor = -14 &= (11110001)_2 \end{aligned}$$

Zweierkomplement:

$$\begin{aligned} -29 &= (11100011)_2 \\ -29 \cdot 2 = -58 &= (11000110)_2 \\ \lfloor -29/2 \rfloor = -14 &= (11110010)_2 \rightarrow \textbf{kein Rechts-Shift!} \end{aligned}$$

Kontrollstrukturen

Kontrollstrukturen → zur Steuerung des Programmablaufs

In C: Kontrollstrukturen für **wohlstrukturierte** Programme

- Zusammenfassen von Anweisungen
- Entscheidungen (`if/else`)
- Auswahl aus einer Menge möglicher Fälle (`switch`)
- Schleifen mit Test des Abbruchkriteriums
  * am Anfang (`while`, `for`)
  * am Ende (`do`)
- vorzeitiges Verlassen einer Schleife (`break`, `continue`) ist nicht wohlstrukturiert!

Zusammenfassen von Anweisungen

Ausdrücke:

```
x = 0
i--
printf(...)
```

→

Anweisungen:

```
x = 0;
i--;
printf(...);
```

Anweisungsfolgen mit geschweiften Klammern zusammenfassen → der geklammerte Block gilt als eine Anweisung

```
{
    x = 0;
    i--;
    printf(...);
}
```

Struktogramm:

Anweisung 1
Anweisung 2
Anweisung 3

Konditional-Ausdrücke

→ werden mittels ternärem Operator `?:` gebildet

Syntax:

```
expr_0 ? expr_1 : expr_2
```

Erklärung: (Semantik)

- Ausdruck `expr_0` auswerten
- falls `expr_0` gilt, dann `expr_1` auswerten, sonst `expr_2`
- es wird entweder `expr_1` oder `expr_2` ausgewertet

Beispiele:

- `max = (i > j) ? i : j;`
- `x = (x < 20) ? 20 : x;`

abweisende/kopfgesteuerte Schleife

Syntax:

```
while (ausdruck)
    anweisung
```

Erklärung: (Semantik)

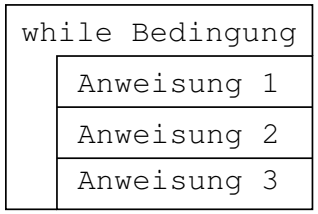
- `anweisung` wird ausgeführt, solange `ausdruck` wahr ist.
- `ausdruck` wird vor jedem Schleifendurchlauf bewertet.
- Kontrollausdruck vorm ersten Durchlaufen der Schleife nicht erfüllt → `anweisung` wird gar nicht durchlaufen

abweisende/kopfgesteuerte Schleife (2)

Beispiel:

```
int i = 0, sum = 0;
while (i < 10) {
    sum += i;
    i++;
}
```

Struktogramm:



fußgesteuerte Schleife

Syntax:

```
do
    anweisung
while (ausdruck);
```

Erklärung: (Semantik)

- Die Anweisung in der Schleife wird ausgeführt, bis der Kontrollausdruck `ausdruck` nicht mehr erfüllt ist.
- **Die Schleifenanweisung `anweisung` wird mindestens einmal ausgeführt!**

fußgesteuerte Schleife (2)

Beispiel:

```
int i = 0, sum = 0;
do {
    sum += i;
    i++;
} while (i < 10);
```

Struktogramm:

do/while vs. repeat/until

In C gibt es keine `repeat/until`-Schleifen. Diese können aber durch `do/while`-Schleifen nachgebildet werden:

```
do                          entspricht      repeat
    .....                                      .....
while B                                     until !B
```

Beispiel:

```
do {                                       repeat {
    sum += i;                                  sum += i;
    i++;                    entspricht         i++;
} while (i < 10);                          } until (i >= 10);
```

Zähl-Schleife

`for`-Schleife: bietet Möglichkeit, einfache Initialisierungen und Zählvorgänge übersichtlich zu formulieren

Syntax: `for (ausdruck_1; ausdruck_2; ausdruck_3)`
`anweisung`

Erklärung: (Semantik)

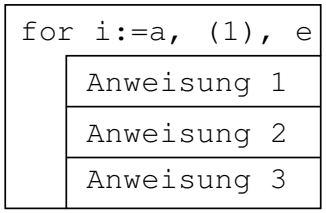
- `ausdruck_1` → Initialisierungsausdruck
- `ausdruck_2` → Abbruchkriterium der Schleife: die Schleife wird solange durchlaufen, wie `ausdruck_2` erfüllt ist
- `ausdruck_3` wird nach jedem Schleifendurchlauf bewertet → Schleifenvariablen ändern

Zähl-Schleife (2)

Beispiel:

```
int i, sum = 0;
for (i = 0; i < 10; i++) {
    sum += i;
}
```

Struktogramm:



Anmerkung: Bei keinem Schleifentyp sind die geschweiften Klammern { und } Bestandteil der Syntax! Gleichbedeutend zu oben:

```
int i, sum = 0;
for (i = 0; i < 10; i++)
    sum += i;
```


Zähl-Schleife (3)

Sollen mehrere Anweisungen bei jedem Schleifendurchlauf ausgeführt werden, dann müssen diese durch { und } zu einer logischen Anweisung (Block) zusammengefasst werden.

Beispiel: Das folgende Programm gibt die Werte n, n^2, n^3, 2^n und $n!$ in Form einer Tabelle aus.

Zähl-Schleife (4)

```
#include <stdio.h>

main() {
    int i, iHoch2, iHoch3;
    int pot2 = 1, fak = 1;

    for (i = 1; i <= 12; i++) {
        iHoch2 = i * i;
        iHoch3 = iHoch2 * i;
        pot2 *= 2;
        fak *= i;
        printf("%2d\t%4d\t%6d\t%8d\t%10d\n",
               i, iHoch2, iHoch3, pot2, fak);
    }
}
```

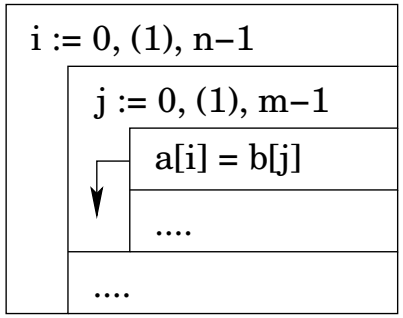
break

Mittels der `break`-Anweisung kann die innerste Schleife vorzeitig und unmittelbar verlassen werden.

Beispiel:

```
for (i = 0; i < n; i++) {
    for (j = 0; j < m; j++) {
        if (a[i] == b[j])
            break;
        .....
    }
    .....
}
```

Struktogramm:



keine strukturierte Programmierung

continue

`continue`-Anweisung: die nächste Wiederholung der umgebenden Schleife wird unmittelbar begonnen.

Beispiel:

```
for (i = 0; i < n; i++) {
    if (a[i] < 0)
        continue;
    .....
}
```

- keine strukturierte Programmierung
- in Struktogrammen nicht darstellbar

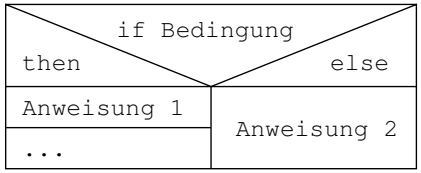
Auswahlanweisungen

Mittels Auswahlanweisungen kann der Ablauf eines Programms abhängig von Bedingungen geändert werden.

Syntax:

```
if (ausdruck)
    anweisung_1
else
    anweisung_2
```

Struktogramm:



Erklärung: (Semantik)

- `ausdruck` wird bewertet. Falls er wahr ist, wird die Anweisung `anweisung_1` ausgeführt, sonst `anweisung_2`.
- Der `else`-Zweig kann entfallen → `anweisung_1` wird übersprungen, falls der Ausdruck `ausdruck` nicht erfüllt ist.

Auswahlanweisungen (2)

In C ist der Wert einer Zuweisung der Wert, der an die linke Seite zugewiesen wird.

- `if (i = 1)` ist immer erfüllt → **Vorsicht!**
- `if (i == 1)` die wahrscheinlich gewünschte Anweisung
- `if (1 == i)` verhindert solche Fehler → **besser!**

Zuweisung kann als Teil eines Ausdrucks verwendet werden!

Beispiele:

- `while ((c = getchar()) != EOF)`
- `if ((l = strlen(s)) > 0)`

⇒ **Probleme beim Debuggen!**

Multiplikations-Algorithmus

```
#include <stdio.h>
main() {
    int a, b, i, n, mult;           /* a, b aus Nat */
    printf("Eingabe: a, b");
    scanf("%d, %d", &a, &b);

    n = 0;              /* Wie viele Stellen hat b? */
    while (b > (1 << n))
        n++;

    mult = 0;                  /* Multiplikation a*b */
    for (i = 0; i < n; i++)
        if (b & (1 << i))
            mult += a << i;
}
```


Multiplikations-Algorithmus (2)

Ablauf: Sei $a = 9 = (1001)_2$ und $b = 13 = (1101)_2$.

Wie viele Stellen hat b?

n	1 << n	b > (1 << n)
0	00001	ja
1	00010	ja
2	00100	ja
3	01000	ja
4	10000	nein

Multiplikation:

i	1 << i	b & (1 << i)	a << i
0	0001	0001	1001
1	0010	0000	----
2	0100	0100	100100
3	1000	1000	1001000

Multiplikations-Algorithmus (3)

C-Freaks schreiben wahrscheinlich:

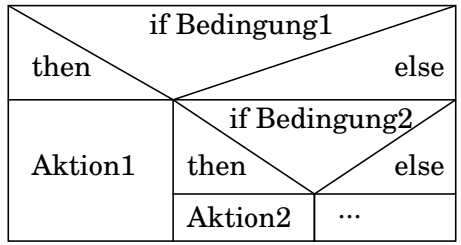
```
#include <stdio.h>
main() {
    int a, b, i, n, mult;
    printf("Eingabe: a, b");
    scanf("%d, %d", &a, &b);

    for (n = 0; b > (1 << n); n++)
        ;

    mult = 0;
    for (i = 0; i <= n; i++)
        (b & (1 << i)) && (mult += a << i);
}
```


Auswahlanweisungen (3)

Syntax:

```
if (ausdruck_1)
    anweisung_1
...
else if (ausdruck_n)
    anweisung_n
else anweisung
```

Struktogramm:



Erklärung: (Semantik)

- `ausdruck_1`, `ausdruck_2`, ... der Reihe nach bewerten
- `ausdruck_i` erfüllt → `anweisung_i` ausführen und die Abarbeitung der Kette abbrechen
- ist kein Ausdruck wahr → die Anweisung im `else`-Teil ausführen. Der `else`-Teil ist optional.

Auswahlanweisungen (4)

Beispiel:
Schaltjahr-Bestimmung mittels Auswahlanweisungen

```
if (jahr % 4 != 0)
    tage = 365;
else if (jahr % 100 != 0)  /* jahr % 4 == 0 */
    tage = 366;
else if (jahr % 400 != 0)  /* jahr % 100 == 0 */
    tage = 365;
else tage = 366;
```


Auswahlanweisungen (5)

Beispiel:
Schaltjahr-Bestimmung mittels Konditional-Ausdruck

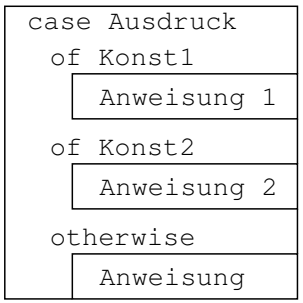
```
tage = (jahr % 4 != 0)
        ? 365
        : ((jahr % 100 != 0)
                ? 366
                : ((jahr % 400 != 0) ? 365 : 366));
```


Auswahl aus mehreren Alternativen

Syntax:

```
switch (ausdruck) {
   case konst_1: anweisung_1
   case konst_2: anweisung_2
   ...
   case konst_n: anweisung_n
   default: anweisung
}
```

Struktogramm:



Erklärung: (Semantik)

1. `ausdruck` wird ausgewertet → muss konstanten ganzzahligen Wert liefern (`switch`-Ausdruck)
2. `case`-Marken werden abgefragt (besteht aus `case`, ganzzahligem konstanten Ausdruck und Doppelpunkt.

`switch`-Anweisung (2)

3. `case`-Konstante stimmt mit `switch`-Ausdruck überein → Programmfluss wird hinter der `case`-Marke fortgesetzt, Anweisungen folgender `case`-Teile werden ausgeführt
4. soll Kette nach Abarbeitung des `case`-Teils beendet werden → explizit durch `break`-Anweisung erzwingen
5. `case`-Konstanten einer Kette müssen unterschiedliche Werte haben.
6. keine übereinstimmende `case`-Marke gefunden → Anweisung hinter der `default`-Marke wird ausgeführt (`default`-Marke und `default`-Anweisung sind optional!)
7. `case`-Konstante darf in einigen Implementierungen keine `const`-Variable sein. (erlaubt: mittels `#define` festgelegte Konstanten.)

`switch`-Anweisung: Beispiel

```
switch (c) {
   case 'm': klein = 1;
   case 'M': n = 1000;
             break;
   case 'd': klein = 1;
   case 'D': n = 500;
             break;
   case 'c': klein = 1;
   case 'C': n = 100;
             break;
   case 'l': klein = 1;
   case 'L': n = 50;
             break;
   case 'x': klein = 1;
   case 'X': n = 10;
             break;
   case 'v': klein = 1;
   case 'V': n = 5;
             break;
   case 'i': klein = 1;
   case 'I': n = 1;
             break;
   default: n = 0;
}
```


Komma-Ausdrücke

Das Komma wird außer als Operator auch als Trennzeichen verwendet, z.B. in Listen von Argumenten.

Die Bedeutung eines Kommas hängt vom Kontext ab, ggf. Komma-Ausdrücke in Klammern setzen, um gewünschtes Ergebnis zu erhalten.

Syntax: `expr_1, expr_2`

Erklärung: zuerst `expr_1` auswerten, dann `expr_2`

Komma-Operator wird dazu verwendet, zwei Ausdrücke an einer Stelle unterzubringen, wo nur ein Ausdruck erlaubt ist, z.B. in Schleifen- oder Funktionsaufrufen.

Komma-Ausdrücke (2)

Beispiel:

```
....
for (i = 0, j = N; i < j; i++, j--) {
    ....
}
a = fkt(x, (y=3, y+2), z);     /* a = fkt(x, 5, z) */
....
```

Typ und Wert des Resultats eines Komma-Ausdrucks sind Typ und Wert des rechten Operanden.

Alle Nebenwirkungen der Bewertung des linken Operanden werden abgeschlossen, bevor die Bewertung des rechten Operanden beginnt.

Format von C-Programmen

C-Programme können formatfrei geschrieben werden, d.h. sie müssen keine bestimmte Zeilenstruktur (im Gegensatz zu Cobol) haben.

Schlüsselwörter und Namen sind getrennt zu schreiben!

Trennzeichen sind:

- Leerzeichen
- horizontaler Tabulator
- vertikaler Tabulator
- Seitenvorschub
- Zeilenende
- Kommentare

In vielen Firmen gibt es Konventionen

- wie Quelltexte zu formatieren sind
- wie Variablen, Tabellen, Dateien usw. zu benennen sind

Format von C-Programmen (2)

guter Einstieg für übersichtliche Formatierung:

`http://java.sun.com/docs/codeconv/index.html`

Ziel unserer Konventionen: größtmögliche

- Deutlichkeit
- Übersichtlichkeit
- Testfreundlichkeit

Im folgenden einige Regeln (Auszüge aus den Java Code Conventions) die für alle C-Programme, die Sie im Rahmen der Veranstaltung *Informatik I* schreiben, verbindlich sind!

Format von C-Programmen (3)

Line length: Avoid lines longer than 80 characters, since they're not handled well by many terminals and tools.

Wrapping lines: When an expression will not fit on a single line, break it according to these general principles:

- Break after a comma.
- Break before an operator.
- Prefer higher-level breaks to lower-level breaks.
- Align the new line with the beginning of the expression at the same level on the previous line.
- If the above rules lead to confusing code or to code that's squished up against the right margin, just indent 8 spaces instead.

Format von C-Programmen (4)

Example: break after a comma

```
someMethod(longExpression1, longExpression2,
        longExpression3, longExpression4,
        longExpression5);
```

Example: prefer higher-level breaks to lower-level breaks

```
var = someMethod1(longExpression1,
                  someMethod2(longExpression2,
                          longExpression3));
```


Format von C-Programmen (5)

Example: breaking an arithmetic expression

```
/* AVOID! */
longName1 = longName2 * (longName3 + longName4
            - longName5) + 4 * longname6;

/* PREFER! */
longName1 = longName2
            * (longName3 + longName4 - longName5)
            + 4 * longname6;
```

Format von C-Programmen (6)

Example: indenting method declarations

```
/* CONVENTIONAL INDENTATION */
someMethod(int anArg, int anotherArg,
           char yetAnotherArg, char andStillAnother) {
    ...
}

/* INDENT 8 SPACES TO AVOID VERY DEEP INDENTS */
long int horkingVeryLongMethodName(int anArg,
        int anotherArg, char yetAnotherArg,
        char andStillAnother) {
    ...
}
```


Format von C-Programmen (7)

Simple Statements: Each line should contain at most one statement.

Blank Lines: Blank lines improve readability by setting off sections of code that are logically related.

- Two blank lines should always be used between sections of a source file.
- One blank line should always be used
  * between methods
  * between the local variables in a method and its first statement
  * before a block or single-line comment
  * between logical sections inside a method

Format von C-Programmen (8)

Blank spaces: should be used in the following circumstances:

- A keyword followed by a parenthesis should be separated by a space.
- A blank space should appear after commas in argument lists.
- All binary operators except . should be separated from their operands by spaces. Blank spaces should never separate unary operators such as unary minus, increment („++“), and decrement („--“) from their operands.
- The expressions in a for statement should be separated by blank spaces.
- Casts should be followed by a blank space.

Format von C-Programmen (9)

Naming conventions: make programs more understandable by making them easier to read.

- Methods should be verbs, in mixed case with the first letter lowercase, with the first letter of each internal word capitalized.

  Example: `run()` oder `getBackgroundColor()`
- The names of variables declared constants should be all uppercase with words separated by underscores.

  Example:

```
#define PI_OVER_2 1.5707963
const int MIN_WIDTH = 4;
```

Format von C-Programmen (10)

Compound statements are statements that contain lists of statements enclosed in braces { statements }.

- The enclosed statements should be indented one more level than the compound statement.
- The opening brace should be at the end of the line that begins the compound statement; the closing brace should begin a line and be indented to the beginning of the compound statement.

Format von C-Programmen (11)

if statements should generally use the 8-space rule, since conventional (4 space) indentation makes seeing the body difficult.

```
/* DON'T USE THIS INDENTATION */
if ((condition1 && condition2)
    || (condition3 && condition4)) {
    doSomethingAboutIt();
}

/* USE THIS INDENTATION INSTEAD */
if ((condition1 && condition2)
        || (condition3 && condition4)) {
    doSomethingAboutIt();
}
```


Kommentare

Jeder Dummkopf kann Code schreiben, den ein Computer versteht. Aber: Gute Programmierer schreiben Code, den Menschen verstehen!

Programme sind grundsätzlich zu kommentieren:

- immer anzugeben: Author, Version, Beschreibung, Datenstrukturen, Rahmenbedingungen, Änderungsliste.
- Code-Blöcke kommentieren, keine einzelnen Zeilen!
- Wenn eine Variable kommentiert werden muss, ist der Name der Variablen schlecht gewählt! → Bezeichner müssen aussagekräftig sein
- Jede Funktion wird beschrieben, dazu gehören u.a. die Parameter und der Rückgabewert.

Kommentare (2)

So nicht:

```
i += 1;                 /* i um eins erhöhen        */
for (j=0; j<=n; j++)    /* zähle j von 0 bis n hoch */
```

sondern so:

```
/* tausche x und y */
if (x > y) {
    x = x - y;
    y = x + y;
    x = y - x;
}
```

oder so:

```
/* berechne s = x*x */
s = 0;
i = 1;
for (j=1; j <= x; j++) {
    s += i;
    i += 2;
}
```

Kommentare (3)

Zur Zeit: Wenn man einen Kommentar benötigt, um zu erklären, was ein Code-Block tut, sollte der Code umgeschrieben werden. → Refactoring: Methode extrahieren

```
...
if (d1.jahr > d2.jahr)
   || (d1.jahr == d2.jahr
      && d1.monat > d2.monat)
   || (d1.jahr == d2.jahr
      && d1.monat == d2.monat
      && d1.tag > d2.tag) {
   ...
}
...
```

```
...
bool greater(date d1,
        date d2) {
    ...
}
...
if (greater(d1, d2)) {
    ...
}
...
```


Ohne Worte

```
#include<stdio.h>
#include<string.h>
main(){char*O,l[999]="’‘acgo\177~|xp .-"
"\OR^8)NJ6%K4O+A2M(*OID57$3G1FBL";while
(O=fgets(l+45,954,stdin)){*l=O[strlen(O)[O-1]=0,
strspn(O,l+11)];while(*O)switch((*l&&isalnum(*O))-!*l){
case-1:{char*I=(O+=strspn(O,l+12)+1)-2,O=34;while
(*I&3&&(O=(O-16<<1)+*I---’-’)<80);putchar(O&93?*I&8||
!(I=memchr(l,O,44))?’?’:I-l+47:32);break;case 1:;}*l=
(*O&31)[l-15+(*O>61)*32];while(putchar(45+*l%2),(*l=
*l+32>>1)>35);case 0:putchar((++O,32));}putchar(10);}}
```

gefunden unter `http://www.arkko.com/ioccc.html`

Die Programmiersprache C

Abgeleitete Datentypen und Funktionen

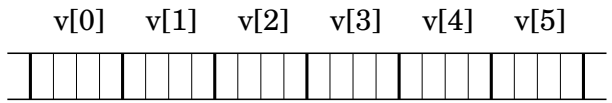
Feldtypen

Zu jedem Typ T kann man ein **Feld von** T erzeugen (man spricht auch von einem **Vektor** oder **Array**).

Bei Feldern handelt es sich um eine Ansammlung von Objekten gleichen Typs, wobei die Objekte in aufeinanderfolgenden Speicherbereichen abgelegt werden.

Ein Element eines Feldes wird über einen Index angesprochen, der in eckigen Klammern angegeben wird.

Beispiel:
`int v[6]` definiert



Feldtypen (2)

Beispiel: Im Folgenden wird ein Feld vom Typ `int` der Größe 10 deklariert. Der Name des Feldes ist `vektor`.

```
int vektor[10];
vektor[0] = 100;
vektor[1] = 2 * vektor[0];
vektor[2] = vektor[0] + vektor[1];
```

Das Array `int a[N]` besitzt die Elemente `a[0]` bis `a[N-1]`.
Vorsicht: Es erfolgt keine Bereichsüberprüfung.

Initialisierung bei der Deklaration:

```
float x1[4] = {1.0, 2.0, 3.0, 4.0};
int   x2[]  = {1, 2, 3, 4};
char  x3[]  = "Hello, world!";
```


Feldtypen (3)

Anmerkungen:

- **Arrays lassen sich nicht mit = zuweisen.**

  Die Anweisung `int a[20];` legt `a` als Name des Arrays fest: Die Inhalte von `a` können verändert werden, nicht `a` selbst.

- **Arrays lassen sich nicht mit == vergleichen.**

  Der Vergleich zweier Arrays ist zwar syntaktisch korrekt, liefert aber in der Regel nicht das gewünschte Ergebnis (siehe Abschnitt „Zeiger und Referenzen“).

Mehrdimensionale Felder

Mehrdimensionale Felder: Anhängen weiterer Indizes.

Beispiel: zweidimensionales Feld `matrix` vom Typ `int`

```
int matrix[10][20];
matrix[0][0] = 42;
matrix[1][0] = matrix[0][0] + matrix[0][1];
```

Mehrdimensionale Arrays und ihre Initialisierung:

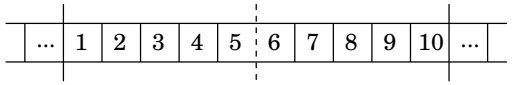
```
float x1[2][4] = { {1.0, 2.0, 3.0, 4.0},
                   {5.0, 6.0, 7.0, 8.0} };
int   x2[][4]   = { {1, 2, 3, 4}, {5, 6, 7, 8} };
char  x3[][7]   = { "Hello,", "world!" };
```


Mehrdimensionale Felder: Anmerkungen

Felder werden zeilenweise abgespeichert. Der letzte Index ändert sich schneller als der erste: `int a[#zeilen][#spalten]`

Beispiel:

```
int a[][5] = {
    {1, 2, 3, 4, 5},
    {6, 7, 8, 9, 10}
};
```



Initialisierung: Nur die erste (äußerste) Größenangabe darf weggelassen werden. Der Rest nicht, da sonst der Index-Operator [] die Position nicht berechnen kann.

Beispiel: Position von `a[i][j]`: `i * #spalten + j`

Aufzählungstyp

Der `enum`-**Typ**: Konstanten werden Integer-Werte zugewiesen, Konstanten sind ansprechbar über Bezeichner.

Beispiel:

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

Die Werte werden intern auf fortlaufende, nicht-negative ganze Zahlen abgebildet, beginnend bei 0. Im Beispiel: `Mo == 0`, `Di == 1`, `Mi == 2` usw.

Eine solche Abbildung kann auch direkt definiert werden:

```
enum Tag {Mo=1, Di=2, Mi=4, Do=8, Fr=16,
          Sa=32, So=64};
```


Aufzählungstyp (2)

Jeder Name darf nur einmal verwendet werden, d.h. Namen in verschiedenen Aufzählungen müssen sich unterscheiden.

Die Werte in einer Aufzählung können gleich sein.

Variablen können mit Aufzählungstypen deklariert werden.

- Sie unterliegen nicht notwendigerweise der Typprüfung.
- Eine Aufzählungskonstante des Typs kann als Wert zugewiesen werden.
- Sie können in logischen Ausdrücken verglichen und in arithmetischen Ausdrücken verknüpft werden.

Aufzählungstyp (3)

Beispiel:

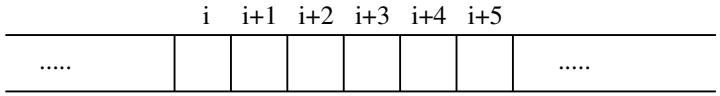
```
enum Tag {Mo, Di, Mi, Do, Fr, Sa, So};
enum Monat {Jan, Feb, Mar, Apr, May, Jun,
        Jul, Aug, Sep, Oct, Nov, Dec};

enum Tag t = Mo;
enum Monat m = Jan;

if (t == Di) ...
if (t == m) ...     /* ist erfüllt: Mo == 0 == Jan */
t = Jan + 3;        /* kein Fehler, keine Warnung  */
```


Zeiger und Adressen

Für jeden Typ T kann man einen **Zeiger auf** T erzeugen. Der Wert eines Zeigertyps ist die Adresse eines Objekts.

Spezieller Wert: der leere Zeiger (NULL-Zeiger) $\rightarrow$ die Konstante 0. Besser: logische Konstante NULL (hat den Wert 0 und ist in `<stddef.h>` definiert).

Vereinfachtes Bild der Speicherorganization:



Speicherzellen sind fortlaufend nummeriert und adressierbar, sie können einzeln oder in zusammenhängenden Gruppen bearbeitet werden.

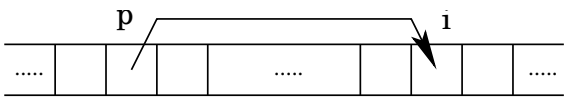
Zeiger und Adressen (2)

Der **Adressoperator** `&` liefert die Adresse eines Objekts.

Der **Inhaltsoperator** `*` liefert das Objekt, das unter einer Adresse abgelegt ist.

Beispiel:

```
int i;      /* Variable für Integer-Wert */
int *p;     /* Zeiger auf einen Integer-Wert */

p = &i;     /* p = Adresse von i: p zeigt auf i */
i = *p;     /* i = Inhalt der Adresse p */
```



Zeiger und Adressen (3)

Die Syntax der Variablenvereinbarung `int *p` imitiert die Syntax von Ausdrücken, in denen die Variable auftreten kann: **Der Ausdruck `*p` ist ein `int`-Wert.**

Daraus folgt:

- Ein Zeiger darf nur auf eine Art von Objekt zeigen.
- Jeder Zeiger zeigt auf einen festgelegten Datentyp.

Ausnahme: Ein **Zeiger auf `void`**

- nimmt einen Zeiger beliebigen Typs auf,
- darf aber nicht selbst zum Zugriff verwendet werden.

Zeiger und Adressen (4)

```
void *v;
int i = 1;
double d = 2.0;

v = &i;                     /* v zeigt auf i */
*(int *) v += 1;            /* cast notwendig: i += 1 */
printf("%d, %d\n", i, *(int *) v);

v = &d;                     /* v zeigt auf d */
*(double *) v += 1.0;       /* cast notwendig: d += 1 */
printf("%f, %f\n", d, *(double *) v);
```


Zeiger und Adressen (5)

Anmerkungen:

- Der Adressoperator `&` kann nur auf Objekte im Speicher angewendet werden, auf Variablen und Vektorelemente.

  Er kann nicht auf Ausdrücke, Konstanten oder `register`-Variablen angewandt werden.

- Die unären Operatoren `*` und `&` haben höheren Vorrang als die arithmetischen Operatoren.

  Beispiele:

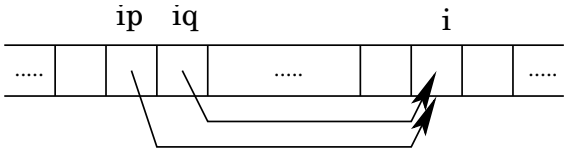
  * `*ip + 10` addiert 10 zu dem Wert, auf den `ip` zeigt.
  * `*ip += 1` inkrementiert den Wert, auf den `ip` zeigt.

Zeiger und Adressen (6)

Anmerkung: Zeiger sind Variablen und können folglich zugewiesen und verglichen werden.

Beispiel:

```
int i, *ip, *iq;

ip = &i;            /* ip zeigt auf i */
iq = ip;            /* iq zeigt auch auf i */
```

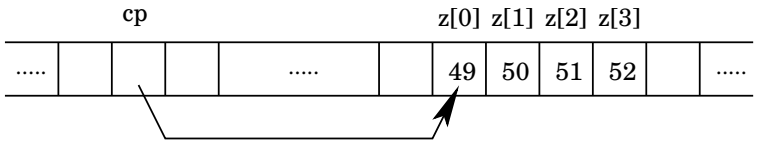


Arrays und Zeiger

Eine Array-Deklaration definiert einen Zeiger auf das erste Element des Arrays.

Beispiel:

```
char *cp;
char z[4] = {'1', '2', '3', '4'};

cp = z;             /* cp zeigt auf z[0] */
```



Arrays und Zeiger (2)

Per Sprachdefinition gilt: Zeigt `p` auf ein Element eines Vektors, dann zeigt `p + j` auf das `j`-te Element hinter `p`.

Vorsicht: Es sind auch negative Werte für `j` zulässig und es findet keine Bereichsüberprüfung statt.

Beispiel:

```
char *cp;
char z[4] = {'1', '2', '3', '4'};

cp = &z[1];         /* cp zeigt auf z[1] */
cp++;               /* cp zeigt auf z[2] */
printf("z[2] = %c, z[-8] = %c\n", *cp, *(cp-10));
```


Arrays und Zeiger (3)

Vorsicht: Arrays lassen sich nicht mit == vergleichen.

Beispiel:

```
int a[3] = {1, 2, 3};
int b[3] = {1, 2, 3};

if (a == b) ...
```

Beispiel:

```
char c1[10], c2[10];

scanf("%s", c1);
scanf("%s", c2);
if (c1 == c2) ...
```

Arrays und Zeiger (4)

Vorsicht: Eindimensionale Arrays können auch als Zeiger deklariert werden, aber

- Zeichenkonstanten liegen im **Programmsegment**,
- Arrays werden im **Datensegment** abgelegt.

Beispiel:

```
char ar[] = "Riker";
char *st = "Worf";

ar[0] = 'X';  /* o.k. */
st = "Troi";  /* o.k. */
st[0] = 'X';  /* nein! */
```

Beispiel:

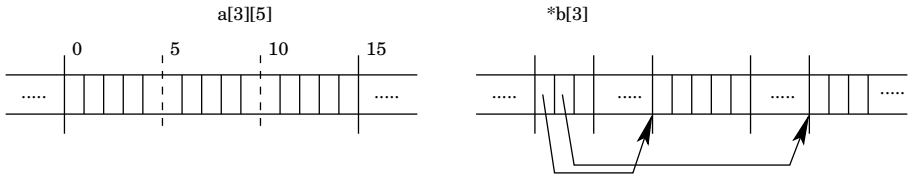
```
char *c1 = "Picard";
char *c2 = "Picard";

if (c1 == c2) ...
```


Zeiger vs. mehrdimensionale Vektoren

Unterscheide 2D-Vektor und Vektor von Zeigern:

```
int a[3][5];
int *b[3];
```

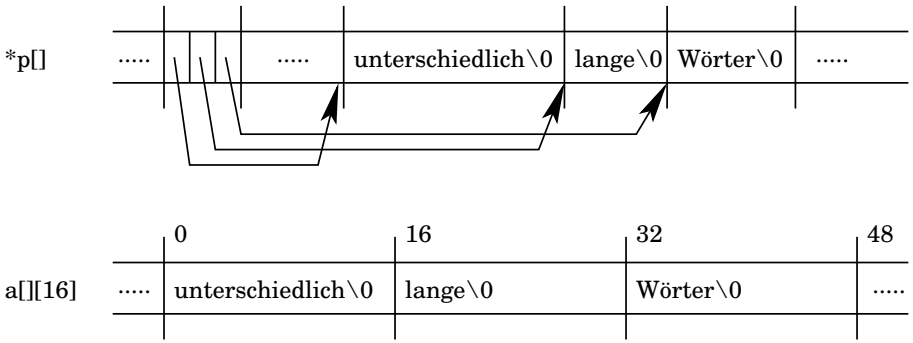
- Legitim: `a[2][4]` und `b[2][4]`.
- Aber `a` ist ein echter 2D-Vektor:
  * `a` stellt für 15 `int`-Werte Speicher bereit.
  * `b` reserviert nur Speicherplatz für 3 Zeiger.



Zeiger vs. mehrdimensionale Vektoren (2)

Oft werden Zeigervektoren benutzt, um Zeichenketten unterschiedlicher Länge zu speichern.

```
char *p[] = {"Unterschiedlich", "lange", "Wörter"};
char a[][16] = {"Unterschiedlich", "lange", "Wörter"};
```



Strukturen

Zusammenfassung von Daten unterschiedlichen Typs.

Syntax:

```
struct <name> {
    <member_1>;
    ....
    <member_n>;
}
```

Beispiel:

```
struct adresse {
    char *str, *ort;
    int hnr, plz;
    long tel;
}
```

Erklärung: (Semantik)

- `<name>` ist der **Typname der Struktur** (kann entfallen)
- `<member_i>` ist eine Variablendeklaration
- die Variablen der Struktur heißen **Komponenten**
- ok: gleiche Komponente in verschiedenen Strukturen

Strukturen (2)

Strukturen ermöglichen die **Organisation von Daten**.

Eine `struct`-Vereinbarung definiert einen Datentyp.

Hinter der Komponentenliste kann eine Liste von Variablen stehen, genau wie bei einem elementaren Datentyp:

Beispiel:

```
int a, b, c;
```

```
struct complex {
    double re, im;
} x, y, z;
```

Beide Definitionen vereinbaren Variablen des angegebenen Typs und reservieren Speicherplatz.

Strukturen (3)

Die Komponenten einer Struktur werden mit dem Punktoperator . angesprochen.

- linker Operand: **Strukturvariable**
- rechter Operand: **Komponentenname**

Beispiel:

```
struct complex {
    double re, im;
} x, y, z;

x.re = 1.0;
x.im = 2.0;
```


Strukturen (4)

Strukturen dürfen andere Strukturen enthalten:

```
struct mitarbeiter {
    char *name, *vorname, *persnr;
    struct adresse adresse;
    int gehalt;
}
```

Strukturen können in Vektoren gespeichert werden:

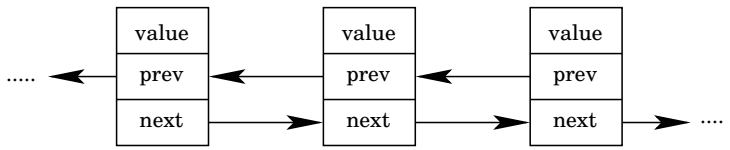
```
struct mitarbeiter personal[20];
...
for (i = 0; i < 20; i++)
    printf("Name[%d] = %s\n", i, personal[i].name);
```


Strukturen (5)

Vorsicht: Strukturen dürfen sich nicht selbst enthalten!
→ wieviel Speicherplatz soll bereitgestellt werden?

Rekursive Strukturen: In der Strukturdeklaration wird ein Zeiger auf die Struktur selbst vereinbart. **Beispiel:**

```
struct liste {
    struct mitarbeiter value;
    struct liste *prev, *next;
} fischer, meier, schulze;
```



Strukturen (6)

Abkürzende Schreibweise für Zeiger auf Strukturen:
`(*name).komponente` entspricht `name->komponente`

Beispiel:

```
fischer.next = &meier;
fischer.prev = 0;
fischer.value.name = "Fischer";
...
schulze.next = 0;
schulze.prev = &meier;
schulze.value.name = "Schulze";
for (l = &fischer; l != NULL; l = l->next)
    printf("%s\n", l->value.name);
```


Strukturen (7)

`typedef`: weise einer Struktur ein **Synonym** zu
→ das Synonym kann genauso genutzt werden, wie ein einfacher Datentyp.

Beispiele:

```
typedef struct {
    char *str, *ort;
    int hnr, plz;
    long tel;
} adresse_t;

adresse_t adr;
adr.str = "Waldweg";
...
```

```
typedef struct {
    char *name, *vname, *pnr;
    adresse_t adresse;
    int gehalt;
} mitarbeiter_t;

mitarbeiter_t arb;
arb.name = "Müller";
...
```


Strukturen (8)

Soll einer rekursiven Struktur mittels `typedef` ein Synonym zugewiesen werden, darf der Strukturname **nicht entfallen**.

Beispiel:

```
typedef struct liste {
    mitarbeiter_t value;
    struct liste *prev, *next;
} liste_t;

liste_t l;
mitarbeiter_t fischer, meier, schulze;
```


Strukturen (9)

Initialisierung: Der Struktur-Definition folgt eine Liste von konstanten Ausdrücken für die Komponenten.

Beispiele:

```
adresse_t a = {"Am Bach", "Aldrup", 4, 12345, 4711};

mitarbeiter_t m = {"Fischer", "Anna", "08F42W",
   {"Am Bach", "Aldrup", 4, 12345}, 2500};

liste_t l = {{"Fischer", "Anna", "08F42W",
   {"Am Bach", "Aldrup", 4, 12345, 4711}}, 0, 0};
```

Anmerkungen:

- Die Reihenfolge der Komponenten ist zu beachten!
- Es müssen nicht alle Komponenten initialisiert werden.

Dynamische Speicheranforderung

Arrays: zur Übersetzungszeit muss bekannt sein, wie viele Elemente gespeichert werden sollen (nicht in C99)

In der Regel ist diese Größe nicht vorhersagbar.
→ Speicher wird zur Laufzeit (dynamisch) angefordert

`void * malloc(size_t size)` liefert Zeiger auf Speicherbereich der Größe `size`, oder `NULL`, wenn die Anfrage nicht erfüllt werden kann. Der Bereich ist nicht initialisiert.

Beispiel:

```
int *pa, i, len;
...
pa = (int *) malloc(len * 4);
if (pa != NULL) ...
```


Dynamische Speicheranforderung (2)

Der unäre Operator `sizeof` liefert die Anzahl der Bytes für ein Objekt oder einen Typ.

Syntax:

```
sizeof <object>
sizeof(type)
```

Beispiel:

```
int i, j;
j = sizeof i;
j = sizeof(int);
```

`sizeof` darf nicht auf Operanden vom Typ Funktion, unvollständige Typen (Felder ohne Größenangabe) oder `void`-Typen angewendet werden.

Beispiel:

```
pa = (int *) malloc(len * sizeof(int));
v = malloc(len * sizeof(int []));   /* falsch! */
```


Dynamische Speicheranforderung (3)

Die Funktion `void free(void *p)` gibt den Speicherbereich, auf den `p` zeigt, wieder frei. Der Speicherbereich muss über einen Aufruf von `malloc` allokiert worden sein!

Beispiel:

```
adresse_t *adr;
...
adr = (adresse_t *) malloc(len * sizeof(adresse_t));
if (adr != 0) {
    adr[0].str = "Waldweg";
    adr[0].hnr = 13;
...
free(adr);
```


Vereinigungstyp

Der **Typ** `union` hat die gleiche Syntax wie der Strukturtyp.

Die einzelnen Komponenten werden **nicht hintereinander**, sondern beginnend an derselben Speicherstelle abgelegt.

Beispiel:

```
union zahlendarstellung {
    unsigned int zahl;
    unsigned char c[4];
} test = {123456};

main() {  .....
    for (i = 0; i < 4; i++)
        printf("Byte %d = %d\n", i, test.c[i]);
}
```

Vereinigungstyp (2)

Die Speicherbereiche überschneiden sich: In der Regel besitzt nur eine Komponente einen sinnvollen Wert.

Beispiel: In einer Symboltabelle sollen die Werte von Programmkonstanten verwaltet werden. Mögliche Typen: `int`, `float` und `char *`.

```
union Value {
    int ival;
    float fval;
    char *cval;
} u;
```

Problem: Der Datentyp, der entnommen wird, muss der Typ sein, der als letzter zuvor gespeichert wurde.

Vereinigungstyp (3)

Der Programmierer muss verfolgen, welcher Typ im `union` gespeichert ist.

Vereinigungstypen können innerhalb von Strukturen und Vektoren auftreten und umgekehrt.

Beispiel:

```
enum SymbolType = {INT, FLOAT, STRING};
struct Symbol {
   char *name;
   enum SymbolType type;
   union Value value;
};
struct Symbol tab[20];
```


Vereinigungstyp (4)

Sicherstellen der Konsistenz ist zum Teil mit erheblichem Aufwand verbunden:

```
for (i = 0; i < 20; i++) {
   printf("%2d: Name = %s, Typ = %d, ",
          i, tab[i].name, tab[i].type);
   if (tab[i].type == INT)
      printf("Wert = %d\n", tab[i].value.ival);
   else if (tab[i].type == FLOAT)
      printf("Wert = %f\n", tab[i].value.fval);
   else printf("Wert = %s\n", tab[i].value.cval);
}
```


Vereinigungstyp (5)

Operatoren:

- zuweisen oder kopieren als Ganzes
- berechnen der Adresse
- Zugriff auf eine Alternative

Syntax: Die Alternativen eines Vereinigungstyps werden angesprochen wie die Komponenten einer Struktur:

```
varname.alternative
varzeiger->alternative
```

Initialisierung: Ein Vereinigungstyp kann nur mit einem Wert initialisiert werden, der zum Typ der ersten Alternative passt (siehe Beispiel Zahlendarstellung).

Funktionen

Funktionen erledigen eine abgeschlossene Teilaufgabe.

Funktionen zerlegen Programme in kleinere Einheiten und erhöhen so die Übersichtlichkeit und die Wartbarkeit von Programmen! → strukturierte Programmierung

Vergleich:

- Funktionen: Programme organisieren
- Strukturen: Daten organisieren

Oft gebrauchte Funktionen und Daten sind in Bibliotheken (Libraries) bereitgestellt: `stdio`, `stdlib`, `string`, `math`, ...

Bei gut entworfenen Funktionen reicht es zu wissen **was** getan wird, gleichgültig **wie** eine Aufgabe gelöst wird.

Funktionen (2)

Syntax:

```
Rückgabetyp Funktionsname( Parameterliste ) {
    Vereinbarungen
    Anweisungen
}
```

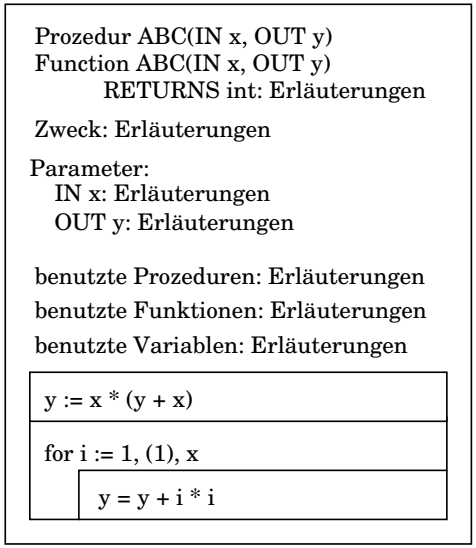
Beispiel:

```
int max(int a, int b) {
    if (a > b)
        return a;
    return b;
}
```

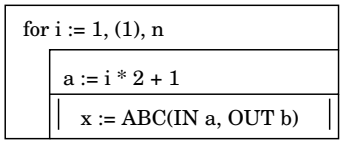
Eine Funktion kann mit `return` einen Wert zurückgeben.

Funktionen: Struktogramm

Definition:



Aufruf:



Funktionen (3)

Gültigkeitsbereich von Bezeichnern:

Parameternamen und **deklarierte lokale Variablen** sind nur innerhalb der Funktion bekannt und für andere Funktionen nicht sichtbar! → andere Funktionen können dieselben Namen ohne Konflikte nutzen!

Rückgabewerte:

- Eine Funktion muss keinen Rückgabewert liefern.
- An der aufrufenden Stelle darf der Rückgabewert einer Funktion ignoriert werden.
- Fehlendes `return`: undefinierter Rückgabewert

Funktionen (4)

Funktionen müssen (wie Variablen) deklariert sein, bevor sie benutzt werden können.

Funktionsdeklaration:

```
int square(int x);
```

Funktionsdefinition:

```
int square(int x) {
    return x*x;
}
```

Funktionen der Standard-Bibliothek werden in **Definitionsdateien** (**header file**) wie `stdio.h` und `math.h` deklariert.

Definitionsdateien werden mittels `#include`-Anweisung am Anfang einer Quelldatei bereitgestellt.

Die Angabe der Parameternamen im Prototyp ist optional.

Funktionen (5)

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

int max(int, int);                  /* Prototyp */

main() {
    int n;
    n = printf("max(%d,%d) = %d\n", 5, 7, max(5, 7));
    printf("#Zeichen in vorheriger Ausgabe: %d\n", n);
    printf("sin(%f) = %f\n", M_PI / 4, sin(M_PI / 4));
}

int max(int a, int b) {             /* Definition */
    return (a > b) ? a : b;
}
```


Funktionen (6)

Rufen sich zwei Funktionen gegenseitig auf, **müssen** zunächst **Funktionsprototypen** definiert werden:

```
int a(int y);    /* Prototyp */
int b(int z);    /* Prototyp */

int a(int x) {
    return b(x - 1) * b(x - 2);
}

int b(int x) {
    return (x <= 0) ? x * 2 : a(x - 10);
}
```

Parameternamen in Prototyp und Funktionskopf müssen **nicht** übereinstimmen.

Funktionen (7)

Compiler-Fehler, wenn Prototyp und Funktionskopf unterschiedliche Rückgabewerte oder Parameter haben.

Beispiel:

```
int sqr(int x);
.....
float sqr(float x) {
    return x*x;
}
```

Beispiel:

```
int fkt(int x, int y);
.....
int fkt(int x) {
    return x*x;
}
```

Aufruf einer Funktion, die vorher **nicht deklariert** wurde:

- Als Rückgabetyp wird `int` angenommen.
- Es werden keine Annahmen über Parameter getroffen.
- Compiler-Verhalten abhängig von der Implementierung.

Funktionen: `main`

In C ist das Hauptprogramm eine Funktion, für die der Name `main` vorgeschrieben ist.

Für jedes ausführbare Programm muss die Funktion `main` existieren, die den Einstiegspunkt bezeichnet.

Damit der Einstiegspunkt eindeutig ist, darf maximal eine Funktion `main` im gesamten Programm existieren.

Im einfachsten Fall definiert man `main` als:

```
main() {
    ....
}
```


Funktionen: `main` (2)

Da für `main` kein Prototyp angegeben wurde, wird automatisch als Rückgabe der Typ `int` angenommen.

Übersetzen des letzten Beispiels liefert Warnungen:

```
warning: return-type defaults to ‘int’
warning: control reaches end of non-void function
```

Soll kein Wert zurückgegeben werden, ist explizit der Rückgabetyp `void` zu definieren.

Struktogramm: analog zu Funktionen und Prozeduren

Funktionen: `main` (3)

Auch `main` kann Parameter von außen übernehmen und Werte zurückgeben.

Definition nach ANSI C:

```
int main(int argc, char *argv[]) {
    ....
    return <ausdruck>;
}
```

Zeichenfolgen als Parameter übergeben:

- `argc` (argument count): Anzahl der beim Aufruf angegebenen Zeichenfolgen (inkl. Programmname).
- `argv` (argument vector): Vektor von Zeigern auf die angegebenen Zeichenfolgen. `argv[0]` = Programmname

Funktionen: `main` (4)

```
#include <stdio.h>

void main(int argc, char *argv[]) {
    int i;
    for (i = 0; i < argc; i++) {
        printf("Parameter %d = %s\n", i, argv[i]);
    }
}
```

Wird das Programm als `arg.c` gespeichert und mit `arg eins 2 drei` ausgeführt, so wird folgende Ausgabe erzeugt:

```
Parameter 0 = arg
Parameter 1 = eins
Parameter 2 = 2
Parameter 3 = drei
```

Funktionen: `main` (5)

Üblich: String-Array zur Abfrage von Umgebungsvariablen.

```
#include <stdio.h>
void main(int argc, char *argv[], char *env[]) {
    int i;
    for (i = 0; env[i] != NULL; i++)
        printf("Variable %d = %s\n", i, env[i]);
}
```

Erzeugt beim Aufruf unter Linux bspw. die Ausgabe:

```
Variable 0 = PWD=/home/jochen
Variable 1 = VENDOR=suse
Variable 2 = PAGER=/usr/bin/less
...
```


Funktionen: `main` (6)

Umwandeln der Programmparameter:
`int sscanf(char *s, ...)`: äquivalent zu `scanf`, aber Eingabezeichen stammen aus Zeichenkette `s`. Rückgabewert: Anzahl der umgewandelten Eingaben.

Beispiel:

```
int i, n, p;
for (i = 0; i < argc; errno = 0, i++) {
   p = sscanf(argv[i], "%d", &n);
   if (p == 0)
      printf("failed to convert %s\n", argv[i]);
   else if (errno == ERANGE)
      printf("value %s out of range\n", argv[i]);
   else printf("%d: value %d\n", i, n);
}
```

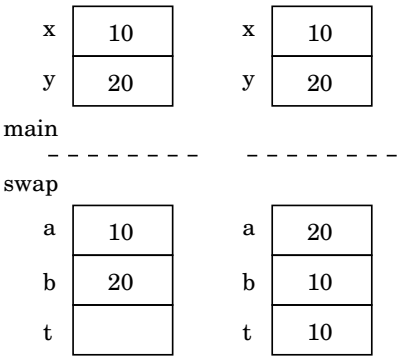

Funktionen: Parameterübergabe

Alle Parameter werden als Wert übergeben (**call by value**).

Innerhalb der Funktionen werden private Kopien der übergebenen Parameter angelegt, die während der Ausführung benutzt werden.

Beispiel:

```
void swap(int a, int b) {
   int t = a;
   a = b;
   b = t;
}
.....
swap(x, y);
```



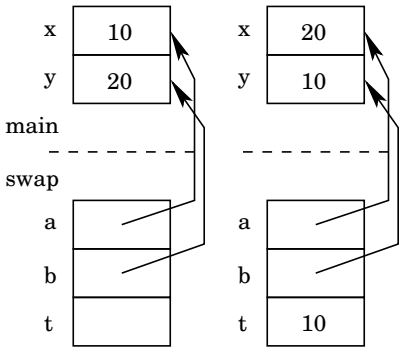
Funktionen: Parameterübergabe (2)

`swap` kann die Werte beim Aufrufer nicht beeinflussen, da nur Werte (Kopien) übergeben werden.

Lösung: Der Aufrufer muss Zeiger auf die Werte, die geändert werden sollen, übergeben (**call by reference**).

Beispiel:

```
void swap(int *a, int *b) {
   int t = *a;
   *a = *b;
   *b = t;
}
.....
swap(&x, &y);
```



Funktionen: Parameterübergabe (3)

Vektoren (Arrays) als Parameter in Funktionen:

- Nur die Adresse des ersten Elements wird übergeben.
- Die Elemente des Vektors werden **nicht** kopiert.
- Es erfolgt **keine** Bereichsüberprüfung.

Beispiel:

```
void main() {
  int i, u[10];
  init(u, 10);
  for (i = 0; i < n; i++)
    printf("%d\n", u[i]);
}
```

```
void init(int *a, int n) {
  for (n--; n >= 0; n--)
    a[n] = n+1;
}
```


Funktionen: Parameterübergabe (4)

Für Zeichenfolgen gilt dasselbe wie für Vektoren:

```
void substitute(char *a, char x, char y) {
   for (; *a != '\0'; a++)
      if (*a == x)
         *a = y;
}
void main() {
   char *s;
   s = (char *) malloc(sizeof("Hallo, Welt!"));
   strcpy(s, "Hallo, Welt!");

   substitute(s, 'l', 'n');
   printf("%s\n", s);
}
```

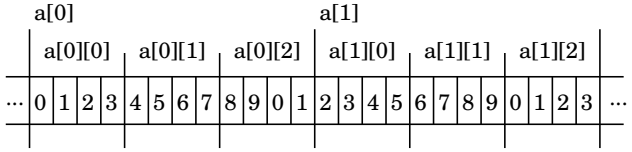

Funktionen: Parameterübergabe (5)

Bei 1D-Vektoren kann die Elementanzahl entfallen.

Bei mehrdimensionalen Vektoren darf nur die Elementanzahl der ersten Dimension weggelassen werden, da sonst der Index-Operator `[]` die Adresse nicht berechnen kann.

Beispiel:

```
type a[2][3][4];
...
b = a[x][y][z];
```



Element `a[x][y][z]` steht an Adresse `a + x*3*4 +y*4 + z`. Für diese Berechnung ist die Elementanzahl der ersten Dimension nicht wichtig.

Funktionen: Parameterübergabe (6)

Zugriff auf 2D-Vektoren mittels Index-Operator:

```
void main() {
    int a[2][4];
    init(a, 2, 4);
    ausgabe(a, 2, 4);
}
void init(int b[][4], int lx, int ly) {
    int i, j, n;
    for (i = 0, n = 0; i < lx; i++)
        for (j = 0; j < ly; j++, n++)
            b[i][j] = n;
}
void ausgabe(int b[][4], int lx, int ly) ....
```

Funktionen: Parameterübergabe (7)

Zugriff auf 2D-Vektoren mittels Zeiger-Arithmetik:

```
void main() {
    int a[2][4];
    init((int *) a, 2, 4);
    ausgabe((int *) a, 2, 4);
}
void init(int *b, int lx, int ly) {
    int i, j, n;
    for (i = 0, n = 0; i < lx; i++)
        for (j = 0; j < ly; j++, n++)
            *(b+i*ly+j) = n;
}
void ausgabe(int *b, int lx, int ly) .....
```


Speicherverwaltung

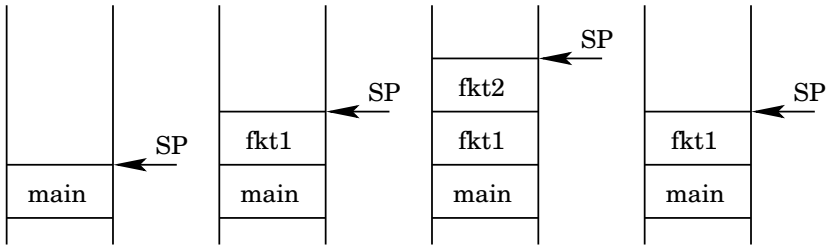
Der von einem C-Programm während seiner Ausführung belegte Speicher ist in vier Bereiche aufgeteilt:

- **Code-Segment**: enthält das übersetzte Programm, also die auszuführenden Maschinenbefehle.
- **Statische Daten**: während des gesamten Lebenszyklus verfügbare Daten wie globale Variablen oder statische, lokale Variablen.
- **Heap:** dynamisch allozierter Speicher (`malloc`)
- **Stack:** Informationen über Funktionsaufrufe

Speicherverwaltung (2)

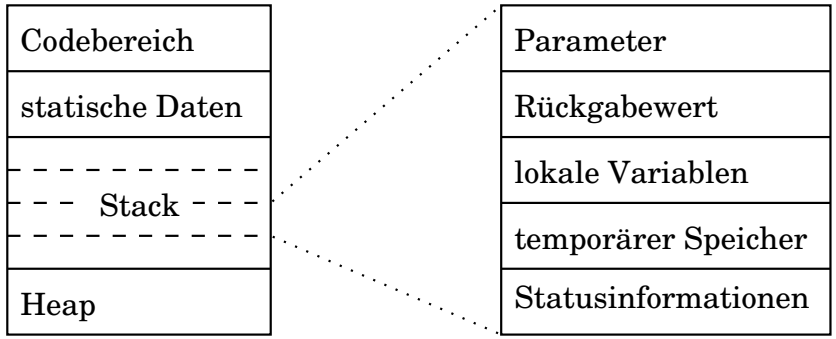
Für die lokalen Variablen und Argumente einer Funktion wird erst beim Aufruf der Funktion Speicherplatz reserviert.

Nach Beenden der Funktion wird der Speicherplatz wieder freigegeben.



Ein CPU-Register (SP: Stack Pointer) enthält die Adresse des nächsten freien Speicherplatzes.

Speicherverwaltung (3)



Alle Informationen, die zum Ausführen einer Funktion notwendig sind, werden in einem **Stack-Frame** abgelegt:

- **temporärer Speicher:** Evaluierung von Ausdrücken
- **Statusinformationen:** Programmzähler, ...

Speicherverwaltung (4)

Vorsicht: Zeiger auf lokale Variablen als Rückgabewert einer Funktion liefern **keinen** definierten Wert!

```
char* intToString(int a) {
    char s[10];
    int i, t;
    for (t = 1; t < a; t *= 10)
        ;
    for (t /= 10, i = 0; t > 0; t/= 10, i++)
        s[i] = (a / t) % 10 + '0';
    s[i] = '\0';
    return s;
}
```


Rekursive Funktionen

Rekursion in der Mathematik:

- **Binomialkoeffizienten:**
$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k} \text{ mit } \binom{n}{n} = 1 \text{ und } \binom{n}{0} = 1$$
- **Fibonacci-Zahlen:**
$$F_n = F_{n-1} + F_{n-2} \text{ mit } F_0 = 0 \text{ und } F_1 = 1$$
- **Determinante:** Die Adjunkte A_{ik} ist die Determinante $(n-1)$ter Ordnung, die durch Streichen der i-ten Zeile und k-ten Spalte, multipliziert mit $(-1)^{i+k}$ entsteht.

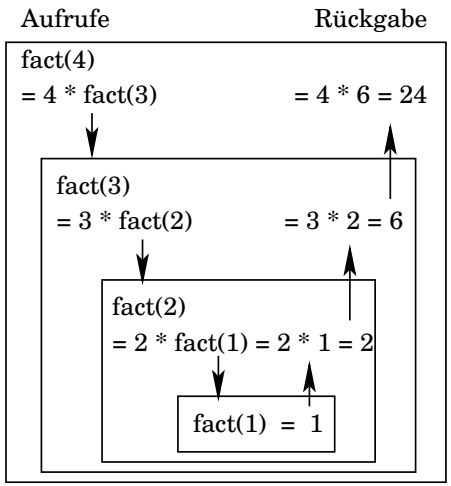
  Entwickeln nach der i-ten Zeile und k-ten Spalte:
$$D(a_{ij}) = \sum_{k=1}^{n} a_{ik} A_{ik}$$

Rekursive Funktionen: Fakultäten

C-Code:

```
#include <stdio.h>
int fact(int n) {
   if (n <= 1)
      return 1;
   return n * fact(n-1);
}
void main() {
   int n;
   for (n = 0; n < 10; n++)
      printf("%d! = %d\n",
             n, fact(n));
}
```

Programmablauf:

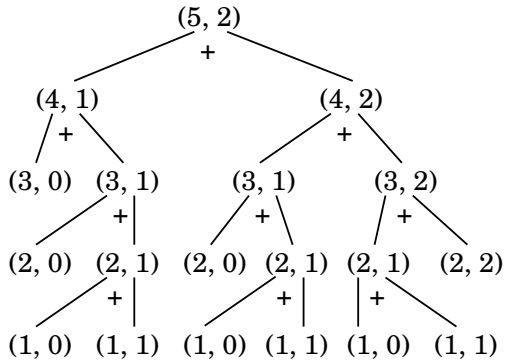


Rekursive Funktionen: Binomialkoeffizienten

C-Code:

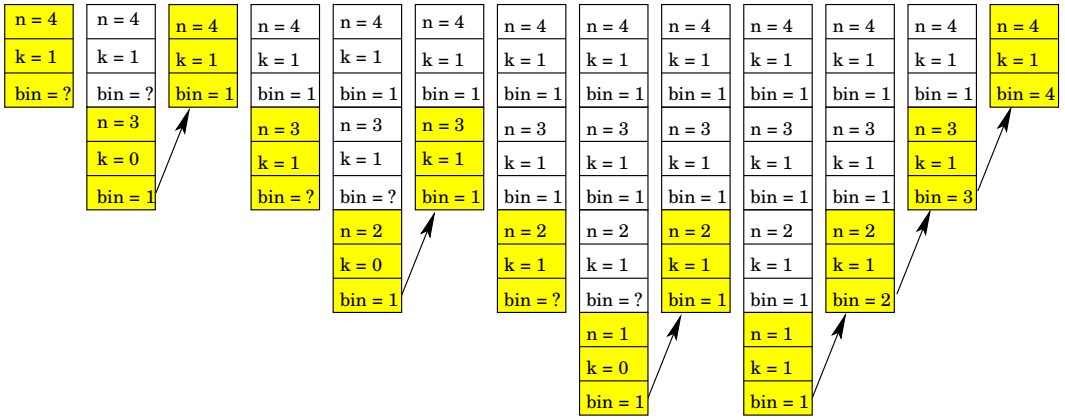
```
#include <stdio.h>
int bin(int n, int k) {
   if (n < k || k < 0)
      return -1;
   if (k == 0 || k == n)
      return 1;
   return bin(n-1, k-1)
      + bin(n-1, k);
}
void main() {
   printf("bin(%d, %d) = %d\n", 7, 3, bin(7, 3));
}
```

Programmablauf:



Rekursive Funktionen: Speicherverwaltung

am Beispiel der Binomialkoeffizienten:



Rekursive Funktionen: Binärdarstellung

```
#include <stdio.h>
void intToBin(int n) {
    if (n < 2) {
        printf("%1d", n);
        return;
    }
    intToBin(n / 2);
    printf("%1d", n % 2);
}

void main(void) {
    intToBin(11);
}
```

Ablauf:

intToBin(n)	Ausgabe
11	
5	
2	
1	1
2	0
5	1
11	1

Gültigkeitsbereich von Bezeichnern

Gültigkeitsbereich (scope): Der Teil des Programms, wo ein Bezeichner benutzt werden kann.

Die Bedeutung einer Deklaration ist abhängig von deren Stelle im C-Code. Mögliche Stellen für Deklarationen:

- Außerhalb von Funktionen: **externe/globale Variablen**
- Im Funktionskopf: **formale Parameter einer Funktion**
- Innerhalb eines Blocks: **lokale Variablen**
- Funktionen können nur außerhalb von Funktionen deklariert werden (parallele Blöcke).

Gültigkeitsbereich von Bezeichnern (2)

Module: C-Quellcode auf mehrere Dateien aufteilen:

- Datei enthält Vereinbarungen und/oder Funktionen.
- Nur eine Datei darf die Funktion `main` enthalten.

Beispiel: Stack

- `stack.h`: enthält die Datenstruktur und die Deklaration aller Zugriffsmethoden
- `stack.c`: Implementierung aller Zugriffsmethoden
- `rechner.c`: Rechner für Postfix-Eingabe (wird auch Umgekehrte Polnische Notation genannt), enthält `main`

Gültigkeitsbereich von Bezeichnern (3)

Es gibt vier mögliche Gültigkeitsbereiche:

- Der Compiler kennt nur, was er bereits gelesen hat, aber keine anderen Dateien. → Externe Variablen und alle Funktionen gelten vom Deklarationspunkt bis zum Ende der Datei.
- Formale Funktionsparameter gelten vom Deklarationspunkt bis zum Ende der Funktion bzw. des Prototypen.
- Lokale Variablen am Beginn eines Blocks gelten bis zum Ende des Blocks.
- Anweisungsmarken gelten innerhalb der Funktion, in der die Deklaration erfolgt.

Gültigkeitsbereich von Bezeichnern (4)

```
/* global: gültig bis zum Ende der Datei */
int global;

/* x: nur innerhalb der Klammern gültig */
double sin(double x);

int main(int argc, char *argv[]) {
    /* lokal: nur innerhalb des Blocks gültig */
    int lokal;
    ...
    /* Label ende: innerhalb der Funktion main gültig */
    ende: ...
}
```


Mehrfachnutzung von Bezeichnern

Derselbe Bezeichner kann gleichzeitig für unterschiedliche Objekte benutzt werden.

Namensklassen: Ein Bezeichner kann auftreten

- als Marke,
- als Name eines Strukturtypen,
- als Komponente innerhalb einer Struktur
- oder als anderer Bezeichner.

Verschiedene Namensklassen können sich überschneiden.

Mehrfachnutzung von Bezeichnern (2)

Beispiel: Namensklassen, die sich überschneiden.

```
int main(int argc, char *argv[]) {
    struct x {
        int x, y;
    } y;
    int x = 5;
    if (x >= 5)
        goto x;
    ...
x:  ...
}
```

Sehr schlechter Programmierstil!

Sichtbarkeit von Bezeichnern

Mehrfachnutzung desselben Bezeichners in einer Namensklasse möglich.

Voraussetzung: Bezeichner tritt in verschiedenen Gültigkeitsbereichen auf.

Beispiel:

```
int x, f;              /* globale Variable */

int main(int cnt, char *args[]) {
    double f;          /* lokale Variable */
    f = sin(1.0);      /* Zugriff auf lokale Var */
    scanf("%d", &x);   /* Zugriff auf globale Var */
    ...
}
```


Sichtbarkeit von Bezeichnern (2)

Die Variablen eines Blocks sind in weiter innen liegenden Blöcken verfügbar, falls keine Variablen mit demselben Bezeichner definiert werden.

Es gelten folgende Sichtbarkeitsregeln:

- Funktionsparameter überschreiben globale Variablen.
- Deklarationen am Anfang eines Blocks überschreiben Deklarationen außerhalb des Blocks.

Lebensdauer von Objekten

Wie lange bleibt ein Objekt im Speicher und kann über seinen Bezeichner angesprochen werden?

Es sind drei Fälle zu unterscheiden:

- **statische Lebensdauer:** Objekte sind stets verfügbar (Funktionen, globale Variablen, statische Variablen).
- **automatische Lebensdauer:** Objekte, die zu Beginn eines Blocks/einer Funktion angelegt werden, sind nach dem Verlassen des Blocks/der Funktion nicht mehr verfügbar, sofern nicht anders definiert.
- **dynamische Lebensdauer:** Der Speicherbereich wird mittels Bibliotheksfunktionen angelegt und freigegeben (`malloc`, `free`).

Lebensdauer von Objekten: Beispiel

```
void fkt(int x) {
    static int s = 1;   /* statische Lebensdauer */
    auto int a = 1;     /* automatische Lebensdauer */
    printf("x = %2d, s = %2d, a = %2d\n", x, s++, a++);
}
void main(void) {
    int i, *p;
    p = (int *) malloc(10 * sizeof(int));
    for (i = 0; i < 10; i++) {
        p[i] = i + 1;   /* dynamische Lebensdauer */
        fkt(p[i]);
    }
    free(p);
}
```

Speicherklasse

Die Speicherklasse eines Objekts hat Einfluss auf

- die Lebensdauer und
- den Gültigkeitsbereich des Objekts und
- erlaubt Einschränkungen für den Zugriff auf Daten.

In C gibt es vier Speicherklassen:

- `auto`
- `register`
- `static`
- `extern`

Speicherklasse: `auto`

Ist nur am Anfang von Blöcken/Funktionen erlaubt.

Definierter Standard:

- alle innerhalb eines Blocks/einer Funktion deklarierten Variablen sind implizit von der Speicherklasse `auto`
- wird in Deklarationen oft weggelassen

Lebensdauer: wird bei jedem Blockeintritt erzeugt und initialisiert, wird nach Verlassen des Blocks wieder gelöscht

Gültigkeit: bis zum Ende des umschließenden Blocks `{ }`

Speicherklasse: `register`

Hinweis an den Compiler, dass auf dieses Objekt oft zugegriffen wird. Soll daher in einem schnellen CPU-Register gespeichert werden. Anzahl prinzipiell unbegrenzt.

Hinweis: Der Compiler ist in keiner Weise daran gebunden.

Verwendung: lokale Variablen und Parameterdeklarationen.

Beispiel:

```
{
    register int i;
    for (i = 0; i < MAX; i++)
        ...
} /* hier wird das Register wieder freigegeben */
```


Speicherklasse: `register` (2)

Lebensdauer: wird bei jedem Blockeintritt erzeugt und nach Verlassen des Blocks wieder gelöscht.

Gültigkeit: bis zum Ende des umschließenden Blocks `{ }`.

Anmerkungen:

- Der Adressoperator `&` ist nicht anwendbar.
- Das Attribut hat keine Bedeutung mehr, da moderne Compiler sehr gute Optimierungsstufen haben.
- In der Praxis: Einschränkungen bzgl. Anzahl und Datentypen, maschinenabhängig.

Speicherklasse: `extern`

Variablen, die außerhalb von Funktionen deklariert werden, sind **implizit** vom Typ `extern`.

Lebensdauer: Für die gesamte Laufzeit des Programms wird Speicherplatz reserviert.

Gültigkeit: Globale Variablen gelten von dem Punkt, an dem sie vereinbart werden, bis zum Ende der Datei.

Ein Objekt einer externen Deklaration ist beim Binden auch Programmteilen aus anderen Dateien bekannt.

Speicherklasse: `extern` (2)

Externe Variable einer Datei in Funktion einer anderen Datei nutzen: Verweis auf Variable angeben.

Beispiel: **schlechter Programmierstil!**

Datei `file1.c`:

```
#include <stdio.h>
int a = 1, b = 3;
int fkt(void);
void main(void) {
    int c = fkt();
    printf("%d, %d, %d\n",
           a, b, c);
}
```

Datei `file2.c`:

```
int fkt(void) {
    extern int a;
    int b = 2;
    return a+b;
}
```


Speicherklasse: `extern` (3)

In der Regel ist Datenaustausch über eine Parameterliste gegenüber globalen Variablen vorzuziehen:

- Global gültige Daten führen oft zu Programmen mit vielen, schwer durchschaubaren Datenpfaden zwischen Funktionen (mit unerwünschten Nebenwirkungen).
- Der Aufbau von Bibliotheken mit allgemein gültigen Funktionen ist nicht möglich.

Weniger Fehleranfällig: Prinzip der Datenkapselung.

- Daten sind nur innerhalb eines Moduls sichtbar.
- Zugriff auf Daten nur über definierte Zugriffsmethoden.
- Beispiel Stack: `push(elem)`, `pop()`, `top()`, `isEmpty()`, ...

Speicherklasse: `static`

Anwendung bei

- **Funktionen:** Die Funktion ist nur innerhalb der Quelldatei und **nicht** dem Linker bekannt (siehe Modulare Programmierung, Datenkapselung).
- **lokale Variablen:** Objekte behalten ihren Wert auch nach Verlassen des Blocks, in dem sie definiert sind.
- **globale Variablen:** Einschränkung des Gültigkeitsbereichs auf die Datei.

Gültigkeit: innerhalb des umschließenden Blocks bzw. bei Funktionen und globalen Variablen innerhalb der Datei.

Speicherklasse: `static` (Beispiel)

Datei `dat1.c`:

```
void f(void) {
    extern int i;
    ...
}

static int i;
int j;
void g(void) {
    i = 42;
    j = 4711;
    ...
}
```

Datei `dat2.c`:

```
void h(void) {
    extern int j;  /* ok */
    extern int i;  /* Fehler! */

    i += 42;
    j += 4711
}
```


Speicherklasse: Beispiel

```
extern int ext, g();      /* global, extern definiert */

static int f(register i) {/* dieser Datei bekannt */
    auto int a;           /* lokale Variable */
    static int x;         /* Wert bleibt erhalten */
    extern int e();       /* lokal, extern definiert */

    for (a = 0; a < 2; a++, i++) x += i * i;
    return e(x);
}
void main(void) {
    register r = ext;     /* lokale Register-Variable */
    printf("f(%d) = %d\n", r, f(r));
    printf("g(%d) = %d\n", r, g(r));
}
```


Pointer Fun

www.cs.stanford.edu/cslibrary/PointerFunCppBig.avi

Testen

Aufgabe: Einlesen einer Datei und umbenennen aller darin enthalten Dateinamen durch Anhängen einer Extension. Der Dateiname sowie die Extension werden als Kommandozeilenparameter an das Programm übergeben.

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

#define LEN 65

int main(int argc, char *argv[]) {
    FILE *file;
    char *filename, *extension, *listfile;
```

Testen (2)

```
    listfile = argv[1];
    extension = argv[2];
    file = fopen(listfile, "r");
    while (fgets(filename, LEN, file) != NULL) {
        char *newFilename;

        strncpy(newFilename, filename, LEN);
        strncat(newFilename, ".", LEN);
        strncat(newFilename, extension, LEN);
        rename(filename, newFilename);
    }

    return 0;
}
```


Testen (3)

Speicherzugriffsfehler! → Fehlerbehandlung einbauen

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

#define LEN 65

int main(int argc, char *argv[]) {
    FILE *file;
    char *filename, *extension, *listfile;

    if (argc != 3) {
        printf("try: %s listfile extension\n", argv[0]);
        return 1;
    }
```


Testen (4)

```
    listfile = argv[1];
    extension = argv[2];
    file = fopen(listfile, "r");
    if (file == NULL) {                          // !!!
        perror(listfile);                        // !!!
        return 1;                                // !!!
    } else printf("file %s opened\n", listfile); // !!!

    while (fgets(filename, LEN, file) != NULL) {
        ...
        if (rename(filename, newFilename) < 0)   // !!!
            perror(filename);                    // !!!
    }
    return 0;
}
```


Testen (5)

Speicherzugriffsfehler → Debug-Ausgaben einbauen

```
    while (fgets(filename, LEN, file) != NULL) {
        char *newFilename;

    #ifdef DEBUG
        printf("filename = %s", filename);
    #endif
        strncpy(newFilename, filename, LEN);
        strncat(newFilename, ".", LEN);
        strncat(newFilename, extension, LEN);
    #ifdef DEBUG
        printf("new filename = %s", newFilename);
    #endif
        ...
```

Testen (6)

Speicherplatz für `filename` und `newFilename` bereitstellen!

```
char filename[LEN];
char newFilename[LEN];
```

keine Datei wurde umbenannt
→ Zeilentrenner `\n` aus Dateinamen entfernen

```
void chomp(char *str) {
    int i = 0;

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


Testen (7)

```
int main(int argc, char *argv[]) {
    ...
    while (fgets(filename, LEN, file) != NULL) {
        char *newFilename;

        chomp(filename);
    #ifdef DEBUG
        printf("filename = %s", filename);
    #endif
        ...
```

einige Dateien wurden nicht umbenannt
→ führende Leerzeichen entfernen

Testen (8)

```
void removeLeadingBlanks(char *str) {
    int i, p = 0;
    int len = strlen(str);

    while (isblank(str[p]))
        p += 1;

    for (i = 0; p < len; i++, p++)
        str[i] = str[p];
    str[i] = '\0';
}
```


Testen (9)

```
int main(int argc, char *argv[]) {
    ...
    while (fgets(filename, LEN, file) != NULL) {
        char *newFilename;

        chomp(filename);
        removeLeadingBlanks(filename);
    #ifdef DEBUG
        printf("filename = %s", filename);
    #endif
        ...
```

Die Programmiersprache C

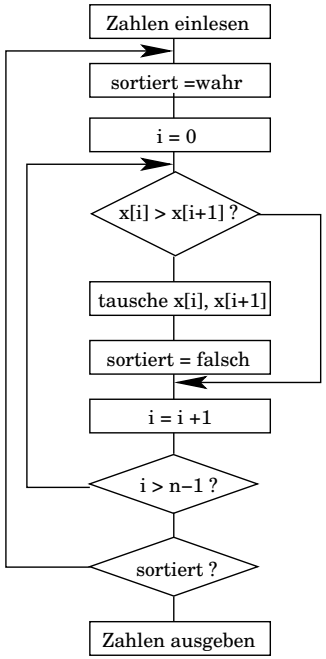
Strukturierte/prozedurale Programmierung

Unstrukturierte Programmierung

Früher verwendete man **Sprünge** in Programmen und **Flussdiagramme** zur Darstellung des Ablaufs.

Nachteil: Sprünge an jede beliebige Stelle erlaubt → **Spaghetti-Code**

In der Folge: Beschränken auf wenige Kontrollstrukturen, Top-Down Entwicklung, verwenden von **Prozeduren** als Strukturmittel.



Prozedurale Programmierung

Was ist das?

- Funktionen und Prozeduren werden dazu benutzt, Programme zu organisieren. (`strToInt`, `getHostByName`, ...)
- Oft gebrauchte Funktionen, Prozeduren und Daten werden in Bibliotheken (Libraries) zur Verfügung gestellt. (`sqrt`, `sin`, `printf`, `scanf`, `malloc`, ...)
- Programmiersprachen unterstützen diesen Stil, indem sie Techniken für die Argumentübergabe an Funktionen und das Liefern von Funktionswerten bereitstellen.
- **Beispiele:** Algol68, FORTRAN, Pascal und C
  Gegenbeispiele: Assembler, Cobol, Basic

Prozedurale Programmierung (2)

Ziele:

- Wartbarer und erweiterbarer Code.
- Wiederverwendung von Algorithmen.
- Wiederverwenden von Code **nicht durch Cut&Paste**, sondern durch allgemeingültige Funktionen.

Probleme:

- **Typisierung:** InsertionSort nur für `int`-Werte?
- **allgemeine Datentypen:** Vergleichsoperatoren?
- **Funktionalität:** was soll passieren, wenn der Benutzer auf eine Schaltfläche (Button) klickt?

Prozedurale Programmierung (3)

Rationale Zahlen:

- öffentliche Operationen:
  * Addition, Subtraktion, Multiplikation, Division
  * Vergleiche: kleiner, größer, gleich
  * Ausgabe
- private Operation: Kürzen (mittels ggT)

Die Implementierung ist im Anwendungsprogramm nicht wichtig, wohl aber die Deklaration der Funktionen.
⇒ aufteilen in die Dateien `rational.h` und `rational.c`

Rationale Zahlen

Datei `rational.h`:

```
typedef struct {
    long int numerator, denominator;
} rational_t;

char *toString(rational_t, char *);
int isEqual(rational_t, rational_t);
int isLess(rational_t, rational_t);

rational_t addR(rational_t, rational_t);
rational_t subR(rational_t, rational_t);
rational_t mulR(rational_t, rational_t);
rational_t divR(rational_t, rational_t);
```


Rationale Zahlen (2)

Datei `rational.c`:

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

char *toString(rational_t r, char *string) {
    sprintf(string, "[%ld/%ld]",
            r.numerator, r.denominator);
    return string;
}
```


Rationale Zahlen (3)

```
void kuerzen(rational_t *rat) {
    long int a = rat->numerator;
    long int b = rat->denominator;

    if (a < 0) a = -a;
    if (b < 0) b = -b;
    while (b > 0) {      /* ggT(a, b) berechnen */
        long int r = a % b;
        a = b;
        b = r;
    }

    rat->numerator /= a;
    rat->denominator /= a;
}
```

Rationale Zahlen (4)

```
rational_t addR(rational_t a, rational_t b) {
    rational_t r;

    r.numerator = a.numerator * b.denominator
                + a.denominator * b.numerator;
    r.denominator = a.denominator * b.denominator;

    kuerzen(&r);
    return r;
}
```


Rationale Zahlen (5)

```
rational_t subR(rational_t a, rational_t b) {
    rational_t r;

    r.numerator = a.numerator * b.denominator
                - a.denominator * b.numerator;
    r.denominator = a.denominator * b.denominator;

    kuerzen(&r);
    return r;
}
```


Rationale Zahlen (6)

```
rational_t mulR(rational_t a, rational_t b) {
    rational_t r;

    r.numerator = a.numerator * b.numerator;
    r.denominator = a.denominator * b.denominator;

    kuerzen(&r);
    return r;
}
```


Rationale Zahlen (7)

```
rational_t divR(rational_t a, rational_t b) {
    rational_t r;

    r.numerator = a.numerator * b.denominator;
    r.denominator = a.denominator * b.numerator;

    kuerzen(&r);
    return r;
}
```

Rationale Zahlen (8)

```
int isLess(rational_t a, rational_t b) {
    rational_t r = sub(a, b);

    if ((r.numerator < 0 && r.denominator > 0)
            || (r.numerator > 0 && r.denominator < 0))
        return 1;
    return 0;
}
int isEqual(rational_t a, rational_t b) {
    rational_t r = sub(a, b);

    if (r.numerator == 0)
        return 1;
    return 0;
}
```


Rationale Zahlen (9)

Datei `main.c`:

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

void main(void) {
    int i;
    char s1[40], s2[40], s3[40];
    rational_t a[] = {{5, 7}, {5, 6}, {3, 7}};
    rational_t b[] = {{3, 4}, {4, 15}, {6, 14}};
    rational_t c;
```


Rationale Zahlen (10)

```
    for (i = 0; i < 3; i++) {
        c = addR(a[i], b[i]);
        printf("%s + %s = %s\n", toString(a[i], s1),
               toString(b[i], s2), toString(c, s3));

        c = mulR(a[i], b[i]);
        printf("%s * %s = %s\n", toString(a[i], s1),
               toString(b[i], s2), toString(c, s3));

        if (isEqual(a[i], b[i]))
            printf("a[%d] == b[%d]\n", i, i);
        else printf("a[%d] != b[%d]\n", i, i);
    }
}
```


Rationale Zahlen (11)

Zugriff auf interne Variablen verhindern:

- in `rational.h` die Strukturvereinbarung entfernen und durch einen unvollständigen Typen ersetzen
- in `rational.c` die aus `rational.h` entfernte Strukturvereinbarung aufnehmen
- in `main.c` die Variablen des unvollständigen Typen durch Zeiger auf `rational_t` ersetzen
- die Parameter- und Rückgabetypen der Funktionen in `rational.h` und `rational.c` als Zeiger auf `rational_t` vereinbaren
- Funktion `create` zum Erzeugen von rationalen Zahlen hinzufügen

Rationale Zahlen überarbeitet

rational.h

```
#ifndef _RATIONAL_H
#define _RATIONAL_H

typedef struct rational rational_t;  /* incomplete */

rational_t *create(long int, long int);
char *toString(rational_t *, char *);
int isEqual(rational_t *, rational_t *);
int isLess(rational_t *, rational_t *);

void addR(rational_t *, rational_t *, rational_t *);
void subR(rational_t *, rational_t *, rational_t *);
void mulR(rational_t *, rational_t *, rational_t *);
void divR(rational_t *, rational_t *, rational_t *);
#endif
```


Rationale Zahlen überarbeitet (2)

rational.c

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

/***** private *****/
typedef struct {
    long int numerator, denominator;
} rational;

/***** public *****/
rational *create(long int n, long int d) {
    rational *r = (rational *) malloc(sizeof(rational));
    r->numerator = n;
    r->denominator = d;
    return r;
}
```


Rationale Zahlen überarbeitet (3)

```
void addR(rational *ret, rational *a, rational *b) {
    ret->numerator = a->numerator * b->denominator
            + a->denominator * b->numerator;
    ret->denominator = a->denominator * b->denominator;

    kuerzen(ret);
}

void mulR(rational *ret, rational *a, rational *b) {
    ret->numerator = a->numerator * b->numerator;
    ret->denominator = a->denominator * b->denominator;

    kuerzen(ret);
}
...
```


Rationale Zahlen überarbeitet (4)

main.c

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

int main(int argc, char *argv[]) {
    int i;
    char s1[40], s2[40], s3[40];
    rational_t *a[3], *b[3], *c;

    a[0] = create(5, 7);
    a[1] = create(5, 6);
    a[2] = create(3, 7);
```

Rationale Zahlen überarbeitet (5)

```
    b[0] = create(3, 4);
    b[1] = create(4, 15);
    b[2] = create(6, 14);
    c = create(1, 1);

    for (i = 0; i < 3; i++) {
        addR(c, a[i], b[i]);
        printf("%s + %s = %s\n", toString(a[i], s1),
               toString(b[i], s2), toString(c, s3));
    ...
}
```


Make-Utility

In größeren Projekten werden im Laufe der Programmierarbeit immer mehr Module fertig und stabil sein.

Um das gesamte Programm zu testen, müssen die Module übersetzt und gelinkt werden. Haben sich seit dem letzten Test Module geändert, müssen sie neu übersetzt und alle davon abhängigen Module müssen neu gelinkt werden.

Das Make-Utility hilft, die Übersicht zu behalten. Das Programm liest eine Datei (`makefile`), die Regeln enthält:

- Wie wird aus den Quelltextdateien das Programm zusammengesetzt und
- von welchen Dateien ist ein Modul abhängig?

Make-Utility (2)

Abhängigkeiten:

- Ausführbares Programm: abhängig von Objektdateien.
- Objektdateien sind abhängig von Quelltextdateien.
- Quelltextdateien: abhängig von den Definitionsdateien (Header-Dateien).

Allgemeine Form des Makefiles:

```
Zieldatei: Dateien, von denen Zieldatei abhängig ist
<tab>  Befehl, um Zieldatei zu erzeugen
Zieldatei: Dateien, von denen Zieldatei abhängig ist
<tab>  Befehl, um Zieldatei zu erzeugen
...
```


Make-Utility: Beispiel

```
OPT=-Wall -ansi -pedantic -g
all: rational.o main.o
       gcc -o calc main.o rational.o
rational.o: rational.c rational.h
       gcc $(OPT) -c rational.c
main.o: main.c rational.h
       gcc $(OPT) -c main.c
```

- Zieldateien werden mit GNU C-Compiler `gcc` erzeugt.
- `make` erzeugt das erste im Makefile angegebene Ziel.
- Der Aufruf `make ziel` erzeugt die angegebene Zieldatei.
- Ziel wird nur dann erzeugt, wenn die Zieldatei älter ist als eine der Dateien, von denen das Ziel abhängig ist.

Sortieren

Viel Rechenzeit entfällt in der Praxis auf Sortiervorgänge.

- Datensätze bestehen nicht aus elementaren, sondern aus zusammengesetzten Datentypen.
- Fragen: Anzahl der Sätze bekannt? Intern oder extern sortieren? Sätze tauschen oder Index berechnen? ...

Datensätze und Schlüssel:

```
typedef struct {
   char *name, *vname;
   long matrikelnr;
   short alter, fb;
} student_t;
```

Sortieren nach:

- `matrikelnr` oder
- `fb, alter` oder
- `name, vname, alter`

Sortieren (2)

Ziel: Sortieren durch Prozedur `sort` verfügbar machen.

Annahmen:

- Nur internes Sortieren (Werte in Vektor gespeichert).
- Anzahl der Datensätze ist bekannt.
- Datensätze werden getauscht, keinen Index anlegen.

Wenn möglich:

- Sortieren nach verschiedenen Schlüsseln ermöglichen.
- Unabhängigkeit von verwendeten Datentypen.

Sortieren: 1. Versuch

```
#include <stdio.h>
...
void main(void) {
    long u[] = {9, 8, 7, 6, 5, 4, 3, 2, 1};
    double v[] = {5.0, 4.0, 3.0, 2.0, 1.0};

    sortLong(u, 9);
    outputLong(u, 9);

    sortDouble(v, 5);
    outputDouble(v, 5);
}
```


Sortieren: 1. Versuch (2)

```
void sortLong(long *a, int n) {
    int i, j;
    long t;
    for (i = 0; i < n; i++)
        for (j = i + 1; j < n; j++)
            if (a[i] > a[j]) {
                t = a[i];
                a[i] = a[j];
                a[j] = t;
            }
}
void outputLong(long *a, int n) ...
```

Sortieren: 1. Versuch (3)

```
void sortDouble(double *a, int n) {
    int i, j;
    double t;
    for (i = 0; i < n; i++)
        for (j = i + 1; j < n; j++)
            if (a[i] > a[j]) {
                t = a[i];
                a[i] = a[j];
                a[j] = t;
            }
}
void outputDouble(double *a, int n) ...
```


Sortieren: 1. Versuch (4)

Probleme:

- Funktionalitäten sind mehrfach implementiert für verschiedene Datentypen:
  - * Sortieren
  - * Ausgabe
  - * Objekte tauschen
- Wiederverwendung nur durch Cut&Paste.

$\Rightarrow$ Fehler sind an vielen Programmstellen zu beseitigen.

Lösung:

- Unabhängigkeit vom Datentyp durch Zeiger auf `void`.
- Objekte tauschen durch Prozedur `swap()` realisieren.

Objekte vertauschen: 1. Versuch

```
void swap(void *a, int x, int y) {
    void t = a[x];
    a[x] = a[y];
    a[y] = t;
}
```

So nicht!

- Der Wert eines Objekts vom Typ `void` kann in keiner Weise verarbeitet werden, er darf weder explizit noch implizit in einen anderen Typ umgewandelt werden.
  Compiler: `variable or field ‘t’ declared void`
- Index-Operator funktioniert bei Zeiger auf `void` nicht.
  Compiler: `warning: dereferencing ‘void *’ pointer`

Objekte vertauschen: 2. Versuch

```
void swap(void *a, void *b) {
    void *t;
    *t = *a;
    *a = *b;
    *b = *t;
}
```

So nicht! Inhaltsoperator funktioniert bei Zeiger auf `void` nicht. Compiler liefert:

```
invalid use of void expression
warning: dereferencing ‘void *’ pointer
```

Objekte vertauschen

Inhalte byte-weise vertauschen:

```
void swap(void *a, int i, int j, int size) {
    char c, *ta = a;
    int k;

    for (k = 0; k < size; k++) {
        c = *(ta + i * size + k);
        *(ta + i * size + k) = *(ta + j * size + k);
        *(ta + j * size + k) = c;
    }
}
```


Reflexion

Was haben wir erreicht?

- gemeinsame Funktionalität `swap` als eine in sich abgeschlossene Funktion bereitgestellt
- `swap` ist unabhängig vom verwendeten Datentyp

Was ist noch zu tun?

- beschränken auf eine Funktion `sort`
- beschränken auf eine Funktion `output`
- sortieren nach verschiedenen Schlüsseln

Sortieren: 2. Versuch

```
...
typedef enum {LONG, DOUBLE, CHAR} type_t;
...
void main(void) {
    char t[] = {'f', 'e', 'd', 'c', 'b', 'a'};
    long u[] = {9, 8, 7, 6, 5, 4, 3, 2, 1};
    double v[] = {6.0, 5.0, 4.0, 3.0, 2.0, 1.0};

    sort(t, 6, CHAR);
    output(t, 6, CHAR);
    sort(u, 9, LONG);
    output(u, 9, LONG);
    sort(v, 6, DOUBLE);
    output(v, 6, DOUBLE);
}
```


Sortieren: 2. Versuch (2)

```
void sort(void *a, int n, type_t type) {
   int i, j;
   for (i = 0; i < n; i++)
      for (j = i + 1; j < n; j++)
         if (type == LONG) {
            if (((long *) a)[i] > ((long *) a)[j])
                swap(a, i, j, sizeof(long));
         } else if (type == CHAR) {
            if (((char *) a)[i] > ((char *) a)[j])
                swap(a, i, j, sizeof(char));
         } else {
            if (((double *) a)[i] > ((double *) a)[j])
                swap(a, i, j, sizeof(double));
         }
}
```

Sortieren: 2. Versuch (3)

```
void output(void *a, int n, type_t type) {
   int i;
   for (i = 0; i < n; i++)
      if (type == LONG)
         printf("a[%d] = %ld\n", i, ((long *) a)[i]);
      else if (type == CHAR)
         printf("a[%d] = %c\n", i, ((char *) a)[i]);
      else printf("a[%d] = %f\n", i, ((double *) a)[i]);
}
```


Sortieren: 2. Versuch (4)

Probleme:

- Lange, unübersichtliche `if/else`-Anweisungen.
- Innerhalb der `if/else`-Konstrukte: Cut&Paste
- Für jeden selbstdefinierten Datentyp (Adresse, Kunde, Vertrag, ...) müssen die `if/else`-Anweisungen erweitert werden.

Lösung:

- Vergleichsoperation pro Datentyp implementieren und an die Sortierfunktion übergeben.

⇒ **Zeiger auf Funktionen**

Zeiger auf Funktionen

Jede Funktion besitzt eine Adresse:

```
int min(int a, int b) { return (a < b) ? a : b; }
int max(int a, int b) { return (a > b) ? a : b; }

void main(void) {
    int (*fp)(int, int);

    fp = &min;          /* fp zeigt auf min */
    printf("min(5, 7) = %d\n", (*fp)(5, 7));

    fp = &max;          /* fp zeigt auf max */
    printf("max(5, 7) = %d\n", (*fp)(5, 7));
}
```


Zeiger auf Funktionen (2)

Erklärung:

- `fp` ist ein Zeiger auf eine Funktion, die einen `int`-Wert liefert und zwei `int`-Werte als Parameter verlangt.
- `*fp` kann für `min` und `max` benutzt werden.
- **nicht verwechseln mit** `int *fp(int, int)`: Funktion, die zwei `int`-Werte als Parameter hat und einen Zeiger auf einen `int`-Wert als Ergebnis liefert!

Zeiger auf Funktionen können

- zugewiesen,
- in Vektoren eingetragen,
- an Funktionen übergeben werden usw.

Zeiger auf Funktionen (3)

Beispiele aus der Standardbibliothek:

- `void qsort(void *base, size_t nmemb, size_t size, int (*compar)(void *, void *))`

  sortiert ein Array mit `nmemb` Elementen der Größe `size` (das erste Element ist bei `base`). Dazu muss eine Vergleichsfunktion `compar` angegeben werden.

- `void *bsearch(void *key, void *base, size_t nmemb, size_t size, int (*compar)(void *, void *))`

  durchsucht ein Array mit `nmemb` Elementen der Größe `size` (erstes Element bei `base`) nach einem Element `key`. Dazu muss eine Vergleichsfunktion `compar` angegeben werden.

Zeiger auf Funktionen (4)

Beispiele aus der Standardbibliothek: (Fortsetzung)

- `void exit(int status)` beendet ein Programm normal. Die `atexit`-Funktionen werden in umgekehrter Reihenfolge ihrer Hinterlegung durchlaufen.
- `int atexit(void (*fcn)(void))` hinterlegt Funktion `fcn`. Liefert einen Wert ungleich 0, wenn die Funktion nicht hinterlegt werden konnte.

Zeiger auf Funktionen (5)

Werden oft bei GUI-Elementen verwendet, wo sie als **callback function** bei einem Benutzer-Event (Ereignis) aufgerufen werden:

- Was soll passieren, wenn der Benutzer auf eine Schaltfläche (Button) klickt?
- Soll das Programm schlafen oder weiterrechnen, wenn das Fenster zum Icon verkleinert werden soll?
- Sind offene Datenbankverbindungen zu schließen, falls der Benutzer das Fenster schließt?

Sortieren: Zeiger auf Funktionen

Was ist zu tun?

- lange, unübersichtliche `if/else`-Anweisungen ersetzen
- Vergleichsfunktionen pro Datentyp implementieren und an die Sortierfunktion als Parameter übergeben
- Ausgabeprozeduren pro Datentyp implementieren und an die Ausgabefunktion als Parameter übergeben

Sortieren

```
...
int main() {
    char t[] = {'f', 'e', 'd', 'c', 'b', 'a'};
    long u[] = {9, 8, 7, 6, 5, 4, 3, 2, 1};
    double v[] = {6.0, 5.0, 4.0, 3.0, 2.0, 1.0};

    sort(t, 6, sizeof(char),
         (int (*)(void *, void *))cmpChar);
    output(t, 6, (void (*)(void *, int))outChar);

    sort(u, 9, sizeof(long),
         (int (*)(void *, void *))cmpLong);
    output(u, 9, (void (*)(void *, int))outLong);
    ...
}
```


Sortieren (2)

```
void sort(void *a, int n, int size,
          int (*cmp)(void *, void *)) {
    int i, j;
    void *x, *y;
    for (i = 0; i < n; i++)
        for (j = i + 1; j < n; j++) {
            x = a + i * size;
            y = a + j * size;
            if ((*cmp)(x, y) > 0)
                swap(a, i, j, size);
        }
}
```


Sortieren (3)

```
int cmpLong(long *x, long *y) {
    if (*x == *y) return 0;
    if (*x > *y) return 1;
    return -1;
}
int cmpDouble(double *x, double *y) {
    if (*x == *y) return 0;
    if (*x > *y) return 1;
    return -1;
}
int cmpChar(char *x, char *y) {
    if (*x == *y) return 0;
    if (*x > *y) return 1;
    return -1;
}
```


Sortieren (4)

```
void output(void *a, int n, void (*out)(void *, int)) {
   int i;

   for (i = 0; i < n; i++) {
      printf("a[%d] = ", i);
      out(a, i);
      printf("\n");
   }
}
```

Sortieren (5)

```
void outChar(char *a, int pos) {
    printf("%c", a[pos]);
}

void outLong(long *a, int pos) {
    printf("%ld", a[pos]);
}

void outDouble(double *a, int pos) {
    printf("%.2f", a[pos]);
}
```


Sortieren (6)

Anstelle von mehreren Sortierprozeduren haben wir jetzt mehrere Vergleichsfunktionen. Warum soll das besser sein?

- die `sort`-Funktionen enthalten mittels Cut&Paste duplizierten Code → fehleranfällig, schlecht erweiterbar
- Für jeden selbstdefinierten Datentyp **müssen** wir
  * eine Vergleichsfunktion und
  * eine Ausgabeprozedur bereitstellen,

  denn:
  * Nach welchem Schlüssel soll sortiert werden?
  * Wie soll die Ausgabe formatiert sein?

Sortieren (7)

Programmiermodelle:

- **modulare Programmierung:** Alle spezifischen Operationen und Daten in einem Modul kapseln.
- **objektorientierte Programmierung:** Module werden zu Klassen. Zusätzlich: Polymorphismus, Vererbung

Reflexion: Wir wollten

- Sortieren durch Prozedur `sort` verfügbar machen,
- Unabhängigkeit von verwendeten Datentypen und
- Sortieren nach verschiedenen Schlüsseln ermöglichen.

der letzte Punkt ist noch nicht erfüllt
⇒ **variabel lange Parameterlisten**

Variabel lange Parameterlisten

Motivation: Wir haben bereits (unbewusst?) variabel lange Parameterlisten angewendet:

```
printf("Ergebnis: %d\n", erg);
printf("fib[%d] = %d\n", i, fib[i]);
printf("bin[%d][%d] = %d\n", i, j, bin[i][j]);

scanf("%d", &no);
scanf("%d, %f", &no, &epsilon);
```

Wie ist das eigentlich realisiert?

Variabel lange Parameterlisten (2)

Motivation: Bei Datenbankzugriffen mittels SQL können die Daten nach verschiedenen Kriterien sortiert werden.

```
select * from student order by matrikelnr;
select * from student order by fachbereich, alter;
select * from student order by name, vorname, alter;
```

Wie sähe denn entsprechender C-Code aus?

```
student_t arr[100];
...
sortBy(arr, "matrikelnr");
sortBy(arr, "fachbereich", "alter");
sortBy(arr, "name", "vorname", "alter");
```


Variabel lange Parameterlisten (3)

Die Standardbibliothek `stdarg` bietet die Möglichkeit, eine Liste von Funktionsargumenten abzuarbeiten, deren Länge und Datentypen nicht bekannt sind.

Beispiele:

```
int printf(const char *format, ...)
int scanf(const char *format, ...)
```

Parameterliste endet mit `...` → die Funktion darf mehr Argumente akzeptieren als Parameter explizit beschrieben sind

`...` darf nur am Ende einer Argumentenliste stehen

Variabel lange Parameterlisten (4)

Beispiel: `int fkt(char *fmt, ...);`

Mit dem **Typ** `va_list` definiert man eine Variable, die der Reihe nach auf jedes Argument verweist.

```
va_list vl;
```

Das **Makro** `va_start` initialisiert `vl` so, dass die Variable auf das erste unbenannte Argument zeigt. **Das Makro muss einmal aufgerufen werden, bevor `vl` benutzt wird.**

Es muss mindestens einen Parameter mit Namen geben, da `va_start` den letzten Parameternamen benutzt, um anzufangen.

```
va_start(vl, fmt);
```


Variabel lange Parameterlisten (5)

Jeder Aufruf des **Makros** `va_arg` liefert ein Argument und bewegt `vl` auf das nächste Argument.

`va_arg` benutzt einen Typnamen, um zu entscheiden, welcher Datentyp geliefert und wie `vl` fortgeschrieben wird.

```
ival = va_arg(vl, int);
sval = va_arg(vl, char *);
```

Vorsicht: Der Typ des Arguments wird nicht automatisch erkannt. Um den korrekten Typ angeben zu können, wird ein Format-String wie bei `printf(const char *, ...)` benutzt.

Variabel lange Parameterlisten (6)

Vorsicht: Das Ende der Liste kann **nicht** anhand eines `NULL`-Wertes erkannt werden.

So nicht!

```
while (vl != NULL) {
    val = va_arg(vl, int);
    ...
}
```

⇒ Die Anzahl und die Datentypen der Parameter müssen bekannt sein. Beides kann mittels eines Format-Strings wie bei `printf` erreicht werden.

Variabel lange Parameterlisten (7)

Beispiel: Format-String

```
for (; *fmt; fmt++) {
   switch(*fmt) {
      case 'd': ival = va_arg(vl, int);    break;
      case 'f': fval = va_arg(vl, double); break;
      case 's': sval = va_arg(vl, char *); break;
   }
   ...
```

Eventuell notwendige Aufräumarbeiten erledigt `va_end`.

```
va_end(vl);
```


Beispiel `stdarg`: Hauptprogramm

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

int fkt(char *, ...);

int main(void) {
    fkt("sfdsd", "eins", 2.0, 3, "vier", 5);
    fkt("fdsd", 6.0, 7, "acht", 9);
    return 0;
}
```


Beispiel `stdarg`: Format-String abarbeiten

```
void getAndPrintNextValue(char c, va_list *l);

int fkt(const char *fmt, ...) {
    int z;
    va_list l;

    va_start(l, fmt);
    for (z = 0; *fmt; fmt++, z++) {
        getAndPrintNextValue(*fmt, &l);
    }
    va_end(l);

    return z;
}
```

Beispiel `stdarg`: Parameter auswerten

```
void getAndPrintNextValue(char c, va_list *l) {
    char *sval;
    int ival;
    double fval;
    if (c == 'd') {
        ival = va_arg(*l, int);
        printf("%d (int)\n", ival);
    } else if (c == 'f') {
        fval = va_arg(*l, double);
        printf("%f (double)\n", fval);
    } else if (c == 's') {
        sval = va_arg(*l, char *);
        printf("%s (char *)\n", sval);
    }
}
```


Sortieren nach verschiedenen Schlüsseln

Wir brauchen

- eine Prozedur `sort` mit variabler Anzahl Parameter
- und eine Vergleichsfunktion für beliebige Schlüssel.

Frage: Warum verwenden wir nicht für jede Kombination von Schlüsseln eine eigene Sortierfunktion?

→ zuviele Kombinationen

$$3 \rightarrow 15 = \binom{3}{1} \cdot 1! + \binom{3}{2} \cdot 2! + \binom{3}{3} \cdot 3!$$
$$4 \rightarrow 64 = \binom{4}{1} \cdot 1! + \binom{4}{2} \cdot 2! + \binom{4}{3} \cdot 3! + \binom{4}{4} \cdot 4!$$
$$5 \rightarrow 325 = \binom{5}{1} \cdot 1! + \binom{5}{2} \cdot 2! + \binom{5}{3} \cdot 3! + \binom{5}{4} \cdot 4! + \binom{5}{5} \cdot 5!$$

→ Code-Duplizierung mittels Cut&Paste

Sortieren nach verschiedenen Schlüsseln (2)

```
int cmpName(student_t *s1, student_t *s2) {
    return strcmp(s1->name, s2->name);
}

int cmpVorname(student_t *s1, student_t *s2) {
    return strcmp(s1->vorname, s2->vorname);
}

int cmpNameVorname(student_t *s1, student_t *s2) {
    int r = strcmp(s1->name, s2->name);
    if (r == 0)
        return strcmp(s1->vorname, s2->vorname);
    return r;
}
...
```


Sortieren nach verschiedenen Schlüsseln (3)

`sort.h`

```
#include <stdarg.h>

void sort(void *liste, int length, int size,
          int (*cmp)(void *liste, int pos1, int pos2,
                     int params, va_list args),
          int params, ...);

void swap(void *liste, int pos1, int pos2, int size);
```

Sortieren nach verschiedenen Schlüsseln (4)

sort.c

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

void swap(void *a, int i, int j, int size) {
    char c, *ta = a;
    int k;

    for (k = 0; k < size; k++) {
        c = *(ta + i * size + k);
        *(ta + i * size + k) = *(ta + j * size + k);
        *(ta + j * size + k) = c;
    }
}
```


Sortieren nach verschiedenen Schlüsseln (5)

```
void sort(void *s, int n, int size,
        int (*cmp)(void *, int, int, int, va_list),
        int params, ...) {
    int i, j;
    va_list l;

    va_start(l, params);
    for (i = 0; i < n; i++)
        for (j = i + 1; j < n; j++) {
            if (cmp(s, i, j, params, l) > 0)
                swap(s, i, j, size);
        }
    va_end(l);
}
```


Sortieren nach verschiedenen Schlüsseln (6)

student.h

```
#include <stdarg.h>

typedef struct {
    char *name, *vname;
    short fb, alter;
    long matrikelnr;
} student_t;

int cmp(student_t *, int pos1, int pos2, int params,
        va_list liste);
void createStudent(student_t *, char *name, char *vname,
        short fb, short alter, long matrikelnr);
void output(student_t);
```


Sortieren nach verschiedenen Schlüsseln (7)

```
void setName(student_t *, char *);
char * getName(student_t);

void setVorname(student_t *, char *);
char * getVorname(student_t);

void setAlter(student_t *, short);
short getAlter(student_t);

void setFB(student_t *, short);
short getFB(student_t);

void setMatrikelnr(student_t *, long);
long getMatrikelnr(student_t);
```

Sortieren nach verschiedenen Schlüsseln (8)

student.c

```
#include "student.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

void setName(student_t *s, char *name) {
    s->name = (char *) malloc(strlen(name) + 1);
    strcpy(s->name, name);
}
char * getName(student_t s) {
    return s.name;
}
```


Sortieren nach verschiedenen Schlüsseln (9)

```
void setVorname(student_t *s, char *vorname) {
    s->vname = (char *) malloc(strlen(vorname) + 1);
    strcpy(s->vname, vorname);
}
char * getVorname(student_t s) {
    return s.vname;
}

void setAlter(student_t *s, short alter) {
    s->alter = alter;
}
short getAlter(student_t s) {
    return s.alter;
}
```


Sortieren nach verschiedenen Schlüsseln (10)

```
void setFB(student_t *s, short fb) {
    s->fb = fb;
}
short getFB(student_t s) {
    return s.fb;
}

void setMatrikelnr(student_t *s, long matrikelnr) {
    s->matrikelnr = matrikelnr;
}
long getMatrikelnr(student_t s) {
    return s.matrikelnr;
}
```


Sortieren nach verschiedenen Schlüsseln (11)

```
void createStudent(student_t *s, char *name, char *vorn,
        short alter, short fb, long matnr) {
    setName(s, name);
    setVorname(s, vorn);
    setAlter(s, alter);
    setFB(s, fb);
    setMatrikelnr(s, matnr);
}

void output(student_t s) {
    printf("%s, %s, FB %d, Alter %d, Matrikelnr %ld\n",
           getName(s), getVorname(s), getFB(s),
           getAlter(s), getMatrikelnr(s));
}
```

Sortieren nach verschiedenen Schlüsseln (12)

```
int cmp(student_t *s, int x, int y, int params,
        va_list list) {
   int r = 0, n = 0;
   char *key;

   for (; r == 0 && n < params; n++) {
      key = va_arg(list, char *);
      if (strcmp(key, "name") == 0)
         r = strcmp(s[x].name, s[y].name);
      else if (strcmp(key, "vorname") == 0)
         r = strcmp(s[x].vorname, s[y].vorname);
      else if (strcmp(key, "fb") == 0)
         r = s[x].fb == s[y].fb
             ? 0
             : (s[x].fb > s[y].fb ? 1 : -1);
```


Sortieren nach verschiedenen Schlüsseln (13)

```
      else if (strcmp(key, "alter") == 0)
         r = s[x].alter == s[y].alter
             ? 0
             : (s[x].alter > s[y].alter ? 1 : -1);
      else if (strcmp(key, "matrikelnr") == 0)
         r = s[x].matrikelnr == s[y].matrikelnr
             ? 0
             : (s[x].matrikelnr > s[y].matrikelnr
                ? 1 : -1);
   }

   return r;
}
```


Sortieren nach verschiedenen Schlüsseln (14)

main.c

```
#include <stdio.h>
#include <stdlib.h>
#include "sort.h"
#include "student.h"

int main(void) {
   int i;
   student_t liste[6], *t = liste;

   createStudent(t++, "Müller" , "Hans", 32, 3, 123456);
   createStudent(t++, "Meier",   "Gabi", 37, 4, 765432);
   createStudent(t++, "Meier",   "Rosi", 34, 4, 987612);
   createStudent(t++, "Müller", "Josef", 38, 2, 471115);
   createStudent(t++, "Müller", "Josef", 35, 6, 121341);
   createStudent(t++, "Maier", "Walter", 32, 3, 213111);
```


Sortieren nach verschiedenen Schlüsseln (15)

```
   printf("unsortiert:\n");
   for (i = 0; i < 6; i++)
       output(liste[i]);

   printf("\nSchlüssel: Name, Vorname, Alter\n");
   sort(liste, 6, sizeof(student_t),
        (int (*)(void *, int, int, int, va_list)) &cmp,
        3, "name", "vorname", "alter");
   for (i = 0; i < 6; i++)
       output(liste[i]);
```

Sortieren nach verschiedenen Schlüsseln (16)

```
    printf("\nSchlüssel: Fachbereich, Alter\n");
    sort(liste, 6, sizeof(student_t),
         (int (*)(void *, int, int, int, va_list)) &cmp,
         2, "fb", "alter");
    for (i = 0; i < 6; i++)
        output(liste[i]);

    printf("\nSchlüssel: Vorname, Alter\n");
    sort(liste, 6, sizeof(student_t),
         (int (*)(void *, int, int, int, va_list)) &cmp,
         2, "vorname", "alter");
    for (i = 0; i < 6; i++)
        output(liste[i]);
    return 0;
}
```


Sortieren: Key-Value-Paare

Funktion `cmp` unabhängig von realer Datenstruktur implementieren → Key-Value-Paare verwenden

types.h:

```
#ifndef _TYPES_H
#define _TYPES_H

typedef enum {
    INT, FLOAT, STRING
} type_t;

typedef union {
    long ival;
    double fval;
    char *sval;
} value_t;
```


Sortieren: Key-Value-Paare (2)

```
typedef struct {
    char *key;
    type_t typ;
    value_t value;
} keyValueElem_t;

typedef struct {
    keyValueElem_t *elem;
    int size;
} keyValue_t;

#endif
```


Sortieren: Key-Value-Paare (3)

sort.h:

```
#include "types.h"
#include <stdarg.h>

int cmp(keyValue_t, keyValue_t, int, va_list);
void sort(keyValue_t *, int, int, ...);
void swap(void *, int, int, int);
keyValueElem_t get(keyValue_t, char *);
```

Sortieren: Key-Value-Paare (4)

sort.c:

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

void swap(void *a, int i, int j, int size) {
    char c, *ta = a;
    int k;

    for (k = 0; k < size; k++) {
        c = *(ta + i * size + k);
        *(ta + i * size + k) = *(ta + j * size + k);
        *(ta + j * size + k) = c;
    }
}
```


Sortieren: Key-Value-Paare (5)

```
keyValueElem_t get(keyValue_t s, char *key) {
    int i = 0;
    char c = 1;
    keyValueElem_t e;

    do {
        e = s.elem[i++];
        c = strcmp(key, e.key);
    } while (i < s.size && c != 0);

    return e;
}
```


Sortieren: Key-Value-Paare (6)

```
int cmp(keyValue_t a, keyValue_t b, int n, va_list l) {
  int r = 0;
  char *key;
  keyValueElem_t kv_a, kv_b;

  for (; r == 0 && n > 0; n--) {
    key = va_arg(l, char*);
    kv_a = get(a, key);
    kv_b = get(b, key);
```


Sortieren: Key-Value-Paare (7)

```
    if (kv_a.typ == INT)
      r = (kv_a.value.ival > kv_b.value.ival
          ? 1
          : kv_a.value.ival < kv_b.value.ival ? -1 : 0);
    else if (kv_a.typ == FLOAT)
      r = (kv_a.value.fval > kv_b.value.fval
          ? 1
          : kv_a.value.fval < kv_b.value.fval ? -1 : 0);
    else r = strcmp(kv_a.value.sval, kv_b.value.sval);
  }
  return r;
}
```

Sortieren: Key-Value-Paare (8)

```
void sort(keyValue_t *s, int n, int p, ...) {
    int i, j;
    va_list l;

    va_start(l, p);
    for (i = 0; i < n; i++) {
        for (j = i + 1; j < n; j++) {
            if (cmp(s[i], s[j], p, l) > 0)
                swap(s, i, j, sizeof(keyValue_t));
        }
    }
    va_end(l);
}
```


Sortieren: Key-Value-Paare (9)

student.h:

```
#include "types.h"

void createStudent(keyValue_t *, char *, char *,
        short, short, long);
void output(keyValue_t);

void setName(keyValue_t *, char *);
char * getName(keyValue_t);

void setVorname(keyValue_t *, char *);
char * getVorname(keyValue_t);
```


Sortieren: Key-Value-Paare (10)

```
void setAlter(keyValue_t *, short);
short getAlter(keyValue_t);

void setFB(keyValue_t *, short);
short getFB(keyValue_t);

void setMatrikelnr(keyValue_t *, long);
long getMatrikelnr(keyValue_t);
```


Sortieren: Key-Value-Paare (11)

student.c:

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

void setName(keyValue_t *s, char *name) {
    s->elem[0].key = "name";
    s->elem[0].typ = STRING;
    s->elem[0].value.sval = name;
}

char * getName(keyValue_t s) {
    return s.elem[0].value.sval;
}
```

Sortieren: Key-Value-Paare (12)

```
void setVorname(keyValue_t *s, char *vname) {
    s->elem[1].key = "vname";
    s->elem[1].typ = STRING;
    s->elem[1].value.sval = vname;
}

char * getVorname(keyValue_t s) {
    return s.elem[1].value.sval;
}
```


Sortieren: Key-Value-Paare (13)

```
void setAlter(keyValue_t *s, short alter) {
    s->elem[2].key = "alter";
    s->elem[2].typ = INT;
    s->elem[2].value.ival = (long)alter;
}

short getAlter(keyValue_t s) {
    return s.elem[2].value.ival;
}
```


Sortieren: Key-Value-Paare (14)

```
void setFB(keyValue_t *s, short fb) {
    s->elem[3].key = "fb";
    s->elem[3].typ = INT;
    s->elem[3].value.ival = (long)fb;
}

short getFB(keyValue_t s) {
    return s.elem[3].value.ival;
}
```


Sortieren: Key-Value-Paare (15)

```
void setMatrikelnr(keyValue_t *s, long matrikelnr) {
    s->elem[4].key = "matrikelnr";
    s->elem[4].typ = INT;
    s->elem[4].value.ival = matrikelnr;
}

long getMatrikelnr(keyValue_t s) {
    return s.elem[4].value.ival;
}
```

Sortieren: Key-Value-Paare (16)

```
void createStudent(keyValue_t *s, char *name,
        char *vname, short alter, short fb,
        long matrikelnr) {
    setName(s, name);
    setVorname(s, vname);
    setAlter(s, alter);
    setFB(s, fb);
    setMatrikelnr(s, matrikelnr);
}

void output(keyValue_t s) {
    printf("%10s, %8s, FB %d, Alter %d, Matrnr %ld\n",
        getName(s), getVorname(s), getFB(s),
        getAlter(s), getMatrikelnr(s));
}
```


Sortieren: Key-Value-Paare (17)

main.c:

```
#include <stdio.h>
#include <stdlib.h>
#include "sort.h"
#include "student.h"

void main(void) {
    int i;
    keyValue_t liste[6], *t = liste;

    for (i = 0; i < 6; i++) {
        liste[i].elem = (keyValueElem_t *)
            malloc(5 * sizeof(keyValueElem_t));
        liste[i].size = 5;
    }
```


Sortieren: Key-Value-Paare (18)

```
createStudent(t++, "Fischer",  "Hans", 32, 3, 1234567);
createStudent(t++, "Meier",    "Gabi", 37, 4, 7654321);
createStudent(t++, "Meier",    "Rosi", 34, 4, 9876123);
createStudent(t++, "Fischer", "Josef", 38, 2, 4711815);
createStudent(t++, "Fischer", "Josef", 35, 6, 1213141);
createStudent(t++, "Maier",  "Walter", 32, 3, 2131411);

printf("unsortiert:\n");
for (i = 0; i < 6; i++)
    output(liste[i]);
```


Sortieren: Key-Value-Paare (19)

```
    printf("\nSchlüssel: Name, Vorname, Alter\n");
    sort(liste, 6, 3, "name", "vname", "alter");
    for (i = 0; i < 6; i++)
        output(liste[i]);

    printf("\nSchlüssel: Fachbereich, Alter\n");
    sort(liste, 6, 2, "fb", "alter");
    for (i = 0; i < 6; i++)
        output(liste[i]);

    printf("\nSchlüssel: Vorname, Alter\n");
    sort(liste, 6, 2, "vname", "alter");
    for (i = 0; i < 6; i++)
        output(liste[i]);
}
```

Sortieren

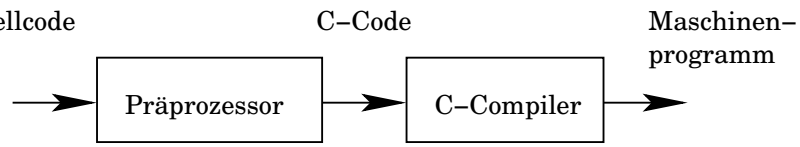
Anmerkungen:

- Die Funktionen `sort()`, `cmp()` und `get()` sind speziell für den Datentyp `keyValue_t` implementiert.
- `sort()` benutzt eine feste Vergleichsfunktion.

Der C-Präprozessor bietet weitere Möglichkeiten, den Code maschinen- und datentyp-unabhängig zu schreiben.

C-Präprozessor

Der Präprozessor bearbeitet den Programm-Code vor der eigentlichen Übersetzung.



Präprozessordirektiven sind Programmzeilen, die mit einem `#`-Zeichen beginnen:

- `#include`: Inhalt einer Datei während der Übersetzung einfügen
- `#define`: Einen Namen durch eine beliebige Zeichenfolge ersetzen (parametrisierbar)

C-Präprozessor: `#include`

Eine Quellzeile wie

`#include <filename>` oder `#include "filename"`

wird durch den Inhalt der Datei `filename` ersetzt:

- `"filename"`: die Suche nach der Datei beginnt dort, wo das Quellprogramm steht.
- `<filename>`: die Datei wird in einem speziellen Verzeichnis gesucht. (Linux: `/usr/include/`)

C-Präprozessor: `#include` (2)

Für den Inhalt der eingefügten Dateien gibt es keine Einschränkungen. Aber: In der Regel werden nur Definitionsdateien eingebunden. Sie enthalten:

- `#define`-Anweisungen,
- weitere `#include`-Anweisungen,
- Typdeklarationen und
- Funktionsprototypen

Die Deklarationen eines großen Programms werden zentral gehalten: Alle Quelldateien arbeiten mit denselben Definitionen und Variablendeklarationen.

C-Präprozessor: `#define`

`#define <name> <ersatztext>` bewirkt, dass im Quelltext die Zeichenfolge `name` durch `ersatztext` ersetzt wird.

Der Ersatztext ist der Rest der Zeile. Eine lange Definition kann über mehrere Zeilen fortgesetzt werden, wenn ein \ am Ende jeder Zeile steht, die fortgesetzt werden soll.

```
#define PRIVATE static
#define PUBLIC
#define ERROR \
    printf("\aEingabedaten nicht korrekt\n");
```

Der Gültigkeitsbereich einer `#define`-Anweisung erstreckt sich von der Anweisung bis ans Ende der Datei.

C-Präprozessor: `#define` (2)

Makros mit Parametern erlauben, dass der Ersatztext bei verschiedenen Aufrufen verschieden sein kann:

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

Ein Makroaufruf ist **kein** Funktionsaufruf! Ein Aufruf von `max` wird **direkt im Programmtext expandiert**. Die Zeile

```
x = MAX(p + q, r + s);
```

wird ersetzt durch die Zeile

```
x = ((p + q) > (r + s) ? (p + q) : (r + s));
```

Textersetzung! Keine Auswertung von Ausdrücken!

C-Präprozessor: `#define` (3)

Hinweis: Anders als bei Funktionen genügt eine einzige Definition von `MAX` für verschiedene Datentypen.

Vorsicht: Im Beispiel werden Ausdrücke eventuell zweimal bewertet. Das führt bei Operatoren mit Nebenwirkungen (Inkrement, Ein-/Ausgabe) zu Problemen.

```
i = 2;
j = 3;
x = MAX(i++, j++);    /* i?? j?? x?? */
```

Vorsicht: Der Ersatztext muss sorgfältig geklammert sein, damit die Reihenfolge von Bewertungen erhalten bleibt:

```
#define SQR(x) x * x
```

Was passiert beim Aufruf `SQR(z + 1)`?

C-Präprozessor: `#define` (4)

Makros können bereits definierte Makros enthalten:

```
#define SQR(x) (x) * (x)
#define CUBE(x) SQR(x) * (x)
```

Die Gültigkeit einer Definition kann durch `#undef` aufgehoben werden:

```
#undef MAX
#undef CUBE
int MAX(int a, int b) ...
```

Textersatz findet nur für Namen, aber nicht innerhalb von Zeichenketten statt.

C-Präprozessor: `#define` (5)

Textersatz und Zeichenketten:

- `#<parameter>` wird durch `"<argument>"` ersetzt.
- Im Argument wird " durch \" und \ durch \\ ersetzt. Resultat: gültige konstante Zeichenkette.

Beispiel: Debug-Ausgabe

```
#define dprint(expr) printf(#expr " = %f\n", expr);
...
dprint(x/y);
/* ergibt: printf("x/y" " = %f\n", x/y); */
```


C-Präprozessor: `#define` (6)

Mittels ## können Argumente aneinandergehängt werden.

Beispiel:

```
#include <stdio.h>
#define paste(head, tail) head ## tail

void main(void) {
    int x1 = 42;
    printf("x1 = %d\n", paste(x, 1));
}
```

Frage: Wozu braucht man das?

C-Präprozessor: `#define` (7)

Auszug aus `stdint.h`:

```
# if __WORDSIZE == 64
#  define __INT64_C(c)  c ## L
#  define __UINT64_C(c) c ## UL
# else
#  define __INT64_C(c)  c ## LL
#  define __UINT64_C(c) c ## ULL
# endif
```

Auszug aus `linux/ext2_fs.h`:

```
#define clear_opt(o, opt)    o &= ~EXT2_MOUNT_##opt
#define set_opt(o, opt)      o |= EXT2_MOUNT_##opt
```


C-Präprozessor: `#if`

Der Präprozessor selbst kann mit bedingten Anweisungen kontrolliert werden, die während der Ausführung bewertet werden:

- Texte abhängig vom Wert einer Bedingung einfügen.
- Programmteile abhängig von Bedingungen übersetzen.

Anwendung:

- um Definitionsdateien nur einmal einzufügen,
- für Debug-Zwecke und
- für systemabhängige Programmteile.

C-Präprozessor: `#if` (2)

`#if <expr>` prüft den Ausdruck (konstant, ganzzahlig).

Ist der Ausdruck „wahr“, werden die folgenden Zeilen bis `else`, `elif` bzw. `endif` eingefügt und damit übersetzt.

Beispiel:

```
#define DEBUG 1
...
#if DEBUG
    printf("Funktion getAdresse(%d): %s\n", i, str);
#endif
```


C-Präprozessor: `#ifdef`

`#ifdef <makro>` bzw. `#ifndef <makro>` prüft, ob ein Makro definiert bzw. nicht definiert ist.

Äquivalente Schreibweise:

```
#if defined <makro>
#if !defined <makro>
```

Beispiel: systemabhängige Übersetzung

```
#ifdef ALPHA
#   include "arch/alpha/semaphore.h"
#elif defined INTEL || defined I386
#   include "arch/i386/semaphore.h"
#endif
```


C-Präprozessor: 1. Beispiel

Jede Definitionsdatei veranlasst das Einfügen derjenigen Definitionsdateien, von denen sie abhängt.

Mehrfaches Einbinden der Datei `netdb.h` verhindern:

```
#ifndef _NETDB_H
#define _NETDB_H 1
/*** eigentlicher Inhalt der Datei netdb.h ***/
#endif
```

Erklärung:

- Beim ersten Einfügen wird `_NETDB_H` definiert.
- Wird die Datei nochmals eingefügt, ist `_NETDB_H` definiert und alles bis zum `#endif` wird übersprungen.

C-Präprozessor: 2. Beispiel

```
#define SWAP(A, B, T) \
{                     \
    T t = A;          \
    A = B;            \
    B = t;            \
}

void sort(long *a, int n) {
    int i, j;
    for (i = 0; i < n; i++)
        for (j = i + 1; j < n; j++)
            if (a[i] > a[j])
                SWAP(a[i], a[j], long)
}
```

Die Programmiersprache C

Die Standardbibliothek

Die Standardbibliothek

Was ist das?

- Sammlung oft benötigter Funktionen/Prozeduren für
  * Ein- und Ausgabe,
  * Tests für Zeichenklassen, String-Bearbeitung,
  * mathematische Funktionen, ...
- Gutes Beispiel für prozedurale Programmierung sowie wiederverwendbarer Software.

Funktionen, Typen und Makros sind in Definitionsdateien deklariert:

`<ctype.h>` `<errno.h>` `<float.h>` `<limits.h>` `<math.h>`
`<stdarg.h>` `<stdio.h>` `<stdlib.h>` `<string.h>` `<time.h>`

Die Standardbibliothek (2)

Definitionsdateien können in beliebiger Reihenfolge und beliebig oft mittels `#include` eingefügt werden.

Eine Definitionsdatei muss außerhalb von allen externen Vereinbarungen eingefügt werden, bevor irgendetwas benutzt wird, das in der Header-Datei vereinbart wird.

Hinweis: Für die Standardbibliothek sind externe Namen reserviert, die mit einem Unterstrich _ und

- einem Großbuchstaben oder
- einem weiteren Unterstrich beginnen.

Ein- und Ausgabe: `stdio.h`

Ein **Datenstrom (stream)**

- ist Quelle oder Ziel von Daten und
- wird mit einem Peripheriegerät verknüpft.

Zwei Arten von Datenströmen werden unterschieden:

- **für Text:** eine Folge von Zeilen, jede Zeile enthält beliebig viele Zeichen und ist mit `\n` abgeschlossen.
- **für binäre Informationen:** eine Folge von Bytes zur Darstellung interner Daten.

Ein- und Ausgabe (2)

Ein Datenstrom wird durch

- **Öffnen (open)** mit der Datei/dem Gerät verbunden,
- **Schließen (close)** von der Datei/dem Gerät getrennt.

Öffnet man eine Datei, erhält man einen Zeiger auf ein Objekt von Typ `FILE`.

Das `FILE`-Objekt enthält alle Informationen, die zur Kontrolle des Datenstroms notwendig sind.

Beim Programmstart sind die Datenströme `stdin`, `stdout` und `stderr` bereits geöffnet.

Ein- und Ausgabe: Dateioperationen

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

`fopen` öffnet die angegebene Datei und liefert einen Datenstrom oder `NULL` bei Misserfolg.

Zu den erlaubten Werten von `mode` gehören:

- `r` zum Lesen öffnen (read)
- `w` zum Schreiben öffnen, alten Inhalt wegwerfen (write)
- `a` zum Anfügen öffnen bzw. erzeugen (append)
- `r+` zum Ändern öffnen
- `w+` zum Ändern erzeugen, alten Inhalt wegwerfen

Wird an die Zeichen ein `b` angehängt, dann wird auf eine binäre Datei zugegriffen, sonst auf eine Textdatei.

Ein- und Ausgabe: Dateioperationen (2)

```
int fflush(FILE *stream)
```

- gepufferte, aber noch nicht geschriebene Daten werden geschrieben
- liefert `EOF` bei einem Schreibfehler, sonst 0

```
int fclose(FILE *stream)
```

- schreibt noch nicht geschriebene Daten
- wirft noch nicht gelesene, gepufferte Daten weg
- gibt automatisch angelegte Puffer frei
- schließt den Datenstrom
- liefert `EOF` bei Fehlern, sonst 0

Ein- und Ausgabe: Dateioperationen (3)

```
int remove(const char *filename)
```

- entfernt die angegebene Datei
- liefert bei Fehlern einen Wert ungleich 0

```
int rename(const char *oldname, const char *newname)
```

- ändert den Namen einer Datei
- liefert im Fehlerfall einen Wert ungleich 0

```
FILE *tmpfile(void)
```

- erzeugt temporäre Datei mit Modus `wb+` (automatisches Löschen bei `fclose` oder normalem Programmende)
- liefert einen Datenstrom oder 0 im Fehlerfall

Formatierte Ausgabe

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

- wandelt die Ausgaben entsprechend `format` um
- schreibt in den angegebenen Datenstrom
- Resultat: Anzahl der geschriebenen Zeichen (negativ im Fehlerfall)

Umwandlungsangabe beginnt mit `%` und enthält optional:

- Steuerzeichen (`%+d`)
- Angabe zur Feldbreite (`%7f`)
- Angabe zur Genauigkeit (`%7.2f`)
- Längenangabe (`%ld`, `%hd`)

Formatierte Ausgabe (2)

Steuerzeichen:

- `-` Ausrichtung linksbündig
- `+` Ausgabe immer mit Vorzeichen
- `0` Auffüllen mit führenden Nullen
- *Leerzeichen* ist das erste Zeichen kein Vorzeichen, so wird ein Leerzeichen vorangestellt
- `#` alternative Form der Ausgabe:
  - `o` die erste Ziffer ist 0
  - `x`, `X` einem Wert wird `0x`/`0X` vorangestellt
  - `e`, `f`, `g` Ausgabe enthält immer einen Dezimalpunkt
  - `g`, `G` Nullen am Ende werden nicht unterdrückt
  - Beispiel `%#X`: hexadezimale Ausgabe anstelle von `%d`

Formatierte Eingabe

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

- liest vom Datenstrom `file` unter Kontrolle von `format`
- legt die umgewandelten Werte in den Argumenten ab
- alle Argumente müssen Zeiger sein
- ist beendet, wenn `format` abgearbeitet ist oder eine Umwandlung nicht durchgeführt werden kann
- liefert `EOF`, wenn vor der ersten Umwandlung das Dateiende erreicht wird oder ein Fehler passiert, ansonsten die Anzahl umgewandelter und abgelegter Eingaben

Formatierte Eingabe (2)

Format-Zeichenkette:

- Leerzeichen und Tabulatorzeichen werden ignoriert: `fscanf(f,"%d %d", &x, &y)` $\widehat{=}$ `fscanf(f,"%d%d", &x, &y)`
- Zeichen, die den nächsten Zeichen nach einem umgewandelten Argument entsprechen müssen: `fscanf(f,"%d,%d", &x, &y)` liest `42,43`, aber nicht `42 43`.
- Umwandlungsangaben: `%` gefolgt von
  - `*`: verhindert Zuweisung an ein Argument (optional)
  - einer Zahl: legt maximale Feldbreite fest (optional)
  - `h`, `l`, `L`: beschreibt die Länge des Ziels; `short`, `long` oder `long double` (optional)
  - einem Umwandlungszeichen

Formatierte Eingabe (3)

Ein Eingabefeld

- ist als Folge von Zeichen definiert, die keine Zwischenraumzeichen sind (Leerzeichen, Tabulator `\t`, Zeilenumbruch `\n`, Seitenvorschub `\f`, ...)
- reicht bis zum nächsten Zwischenraumzeichen, oder bis eine explizit angegebene Feldbreite erreicht ist.

Hinweis: `fscanf` liest über Zeilengrenzen hinweg, um seine Eingabe zu finden.

Formatierte Eingabe: Beispiele

Format	Eingabe	Resultat
`%d%d`	`42 43`	`42 43`
	`42a 43`	`42 %`
	`42 43a`	`42 43`
`%d,%d`	`42,43`	`42 43`
	`42, 43`	`42 43`
	`42, a43`	`42 %`
	`42, 43a`	`42 43`
`%da,%d`	`42a,43`	`42 43`
	`42b,43`	`42 %`

Fehlerbehandlung

Viele der IO-Bibliotheksfunktionen notieren, ob ein Dateiende gefunden wurde oder ein Fehler aufgetreten ist:

- `void clearerr(FILE *file)` löscht die Notizen über Dateiende und Fehler für den Datenstrom `file`.
- `int feof(FILE *file)` liefert einen Wert ungleich Null, wenn für `file` ein Dateiende notiert ist.
- `int ferror(FILE *file)` liefert einen Wert ungleich Null, wenn für `file` ein Fehler notiert ist.

In `errno.h` ist eine global gültige Variable `errno` definiert. Sie enthält eine Fehlernummer, die Informationen über den zuletzt aufgetretenen Fehler zulässt.

Fehlerbehandlung (2)

`void perror(const char *s)` gibt `s` und eine von der Implementierung definierte Fehlermeldung aus, die sich auf den Wert in `errno` bezieht.

```
#include <stdio.h>

void main(void) {
    FILE *f;
    f = fopen("xyz.txt", "r");
    if (f == NULL)
        perror("xyz.txt");
    else fclose(f);
}
```

Wenn die Datei `xyz.txt` nicht gefunden wird, erhalten wir bspw. die Ausgabe: `xyz.txt: No such file or directory`

Ein- und Ausgabe: Beispiel

Ziel: Eine Liste von Studenten in einer Datei abspeichern und aus einer Datei lesen.

Problem: Da die Anzahl der Datensätze in der Datei nicht bekannt ist, implementieren wir zunächst eine lineare Liste, der besseren Übersicht halber in einem eigenen Modul.

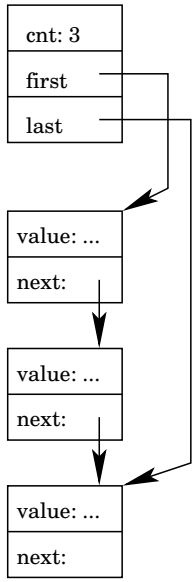
Datei `liste.h:`

```
typedef struct {
    char *name, *vorname;
    short alter, fb;
    long matrikelnr;
} student_t;
```

```
typedef struct elem {
    student_t value;
    struct elem *next;
} listElem_t;
...
```


Ein- und Ausgabe: Beispiel (2)

```
typedef struct {
    int cnt;
    listElem_t *first;
    listElem_t *last;
    listElem_t *pos;
} liste_t;

liste_t *createList(void);
void append(liste_t *l, student_t s);
int getNext(liste_t *l, student_t *s);
void reset(liste_t *l);
```



Ein- und Ausgabe: Beispiel (3)

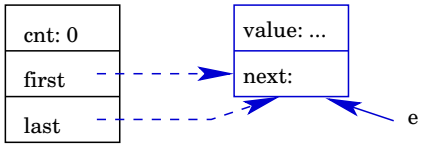
`liste.c:`

```
#include <stdlib.h>
#include "liste.h"

liste_t *createList(void) {
    liste_t *l = (liste_t *) malloc(sizeof(liste_t));
    l->cnt = 0;
    l->first = NULL;
    l->last = NULL;
    l->pos = NULL;
    return l;
}
...
```

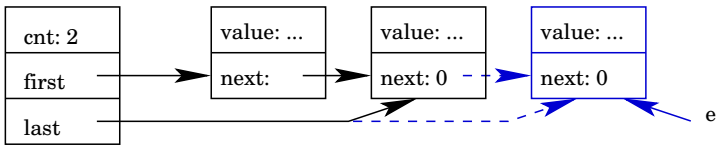

Ein- und Ausgabe: Beispiel (4)

Wenn ein neues Element an die Liste angehängt werden soll, sind zwei Fälle zu unterscheiden:

1. Fall: `cnt == 0`



2. Fall: `cnt > 0`



Ein- und Ausgabe: Beispiel (5)

```
void append(liste_t *l, student_t s) {
    listElem_t *e;
    e = (listElem_t *) malloc(sizeof(listElem_t));
    e->value = s;
    e->next = NULL;
    if (l->cnt == 0) {
        l->first = l->last = l->pos = e;
    } else {
        l->last->next = e;
        l->last = e;
    }
    l->cnt += 1;
}
```


Ein- und Ausgabe: Beispiel (6)

```
int getNext(liste_t *l, student_t *s) {
    if (l->pos == NULL)
        return 0;

    *s = l->pos->value;
    l->pos = l->pos->next;
    return 1;
}

void reset(liste_t *l) {
    l->pos = l->first;
}
```


Ein- und Ausgabe: Beispiel (7)

student.c:

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

void createStudent(student_t *s, char *name,
        char *vorname, short alter, short fb,
        long matrikelnr) {
    s->name = name;
    s->vorname = vorname;
    s->alter = alter;
    s->fb = fb;
    s->matrikelnr = matrikelnr;
}
...
```


Ein- und Ausgabe: Beispiel (8)

```
int saveToFile(student_t *s, int n, char *filename) {
    int i;
    FILE *f;

    f = fopen(filename, "w");
    if (f == NULL)
        return -1;

    for (i = 0; i < n; i++)
        fprintf(f, "%s,%s,%hd,%hd,%08ld\n",
            s[i].name, s[i].vorname, s[i].alter,
            s[i].fb, s[i].matrikelnr);

    return (fclose(f) == EOF ? -2 : 0);
}
```

Ein- und Ausgabe: Beispiel (9)

```
liste_t *readFromFile(char *filename) {
    liste_t *l;
    student_t *s;
    FILE *f;
    char *name, *vorname;
    short alter, fb;
    long nr;

    f = fopen(filename, "r");
    if (f == NULL)
        return NULL;

    l = createList();
    ...
```


Ein- und Ausgabe: Beispiel (10)

```
    name = (char *) malloc(20 * sizeof(char));
    vorname = (char *) malloc(20 * sizeof(char));
    while (fscanf(f, "%5s,%4s,%hd,%hd,%ld",
                name, vorname, &alter, &fb, &nr) == 5) {
        s = (student_t *) malloc(sizeof(student_t));
        createStudent(s, name, vorname, alter, fb, nr);
        append(l, *s);
        name = (char *) malloc(20 * sizeof(char));
        vorname = (char *) malloc(20 * sizeof(char));
    }

    fclose(f);
    return l;
} /* End: readFromFile() */
```


Ein- und Ausgabe: Beispiel (11)

```
void main(void) {
   liste_t *l;
   student_t s[3], *p = s, t;

   createStudent(p++, "Huber", "Hans", 42, 3, 12345678);
   createStudent(p++, "Mayer", "Gabi", 37, 4, 7654321);
   createStudent(p++, "Meier", "Rosi", 34, 4, 98761234);

   saveToFile(s, 3, "_student.txt");
   l = readFromFile("_student.txt");

   while (getNext(l, &t))
      printf("%s, %s, %hd, %hd, %08ld\n", t.name,
             t.vorname, t.alter, t.fb, t.matrikelnr);
}
```


Ein- und Ausgabe: Fragen/Übungen

Es sollen Namen und Vornamen abgelegt werden, die beliebig lang sind (maximal 50 Zeichen). Es ist auch mehr als ein Vorname möglich. Die einzelnen Teile (Name, Vorname, Alter, FB, Matrikelnummer) sollen durch Kommata getrennt sein.

- Welches Problem tritt auf, wenn mit `fscanf` einzelne Strings eingelesen werden? Wie müsste die Funktion `readFromFile` umgeschrieben werden, um CSV-Dateien (Comma Separated Values) einzulesen?
- Schreiben Sie die `getNext`-Funktion so um, dass ein Zeiger auf `Student` zurückgegeben wird, oder Null bei Fehler. Wie ist die Funktion `main` umzuschreiben?

Antwort

Einlesen mittels

```
fscanf(file, "%s,%s,%hd,%hd,%ld", name, vname, ...)
```

funktioniert nicht:

- Beim Format `%s` wird das Komma mitgelesen und kann daher nicht als Trennsymbol verwendet werden.

  Einlesen von `Huber, Hans Walter, 42, 3, 08154711` wird als Namen `Huber,` liefern, anschließendes Komma wird nicht gefunden.
- Bei fester Feldbreite/Längenbeschränkung, bspw. `%50s` wird ggf. nur ein Vorname geliefert, da das Einlesen bei einem Leerzeichen abgebrochen wird.

Lösung: `readFromFile`

```
char teil[50];
...
while (fscanf(file, "%s", name) != EOF) {
    name[strlen(name) - 1] = '\0';  // Komma abschneiden
    vorname[0] = '\0';
    do {
        fscanf(file, "%s", teil);
        strcat(vorname, teil);    // concatenate strings
    } while (teil[strlen(teil) - 1] != ',');
    vorname[strlen(vorname) - 1] = '\0';
    fscanf(file, "%hd,%hd,%ld", &alter, &fb, &matr);
    ...
}
```


Lösung: `getNext`

```
student_t *getNext(liste_t *l) {
    student_t *s;

    if (l->pos == NULL)
        return NULL;

    s = &(l->pos->value);
    l->pos = l->pos->next;
    return s;
}
```


Lösung: `main`

```
void main(void) {
   liste_t *l;
   student_t s[3], *p = s;

   createStudent(p++, "Huber", "Hans", 42, 3, 12345678);
   createStudent(p++, "Mayer", "Gabi", 37, 4, 7654321);
   createStudent(p++, "Meier", "Rosi", 34, 4, 98761234);

   saveToFile(s, 3, "_student.txt");
   l = readFromFile("_student.txt");

   while ((p = getNext(l)) != NULL)
      printf("%s, %s, %hd, %hd, %08ld\n", p->name,
            p->vorname, p->alter, p->fb, p->matrikelnr);
}
```

Ein- und Ausgabe von Zeichen

`int fgetc(FILE *file)`

- liefert das nächste Zeichen des Datenstroms als `unsigned char` (umgewandelt in `int`)
- im Fehlerfall oder bei Dateiende `EOF`

`char *fgets(char *s, int n, FILE *file)`

- liest höchstens die nächsten `n-1` Zeichen in `s` ein
- hört vorher auf, wenn ein Zeilentrenner gefunden wird
- der Zeilentrenner wird im Vektor abgelegt
- der Vektor wird mit `\0` abgeschlossen
- liefert 0 bei Dateiende oder im Fehlerfall, ansonsten `s`

Ein- und Ausgabe von Zeichen (2)

`int fputc(int c, FILE *file)`

- schreibt das Zeichen `c` (umgewandelt in `unsigned char`) in den Datenstrom
- liefert das ausgegebene Zeichen, im Fehlerfall `EOF`

`int fputs(const char *s, FILE *file)`

- schreibt die Zeichenkette `s` in den Datenstrom
- liefert einen nicht-negativen Wert, im Fehlerfall `EOF`

Ein- und Ausgabe von Zeichen: Beispiel

Häufigkeit der Buchstaben in einem Text bestimmen.

```
#include <stdio.h>

int getIndex(char c) {
    /* aus Klein- mach Großbuchstabe */
    if (c >= 'a' && c <= 'z')
        c -= 'a' - 'A';
    /* alle anderen Zeichen ausschließen */
    if (c < 'A' || c > 'Z')
        return -1;
    /* A..Z auf 0..25 abbilden */
    return c - 'A';
}
```


Ein- und Ausgabe von Zeichen: Beispiel (2)

```
int main(int argc, char *argv[]) {
    int i, H[26] = {0};
    char c, *filename = "_test.txt";
    FILE *file;

    if (argc > 1)
        filename = argv[1];
    printf("untersuche Datei %s!\n", filename);

    if (!(file = fopen(filename, "r"))) {
        perror(filename);
        return -1;
    }
    ...
```

Ein- und Ausgabe von Zeichen: Beispiel (3)

```
    while ((c = fgetc(file)) != EOF) {
        if ((i = getIndex(c)) >= 0)
            H[i] += 1;
    }

    fclose(file);

    for (i = 0, c = 'a'; c <= 'z'; i++, c++)
        printf("H[%c] = %d\n", c, H[i]);
    return 0;
}
```


Direkte Ein- und Ausgabe

```
size_t fread(void *ptr, size_t size, size_t nobj,
             FILE *file)
```

- liest aus dem Datenstrom in den Vektor `ptr` höchstens `nobj` Objekte der Größe `size` ein
- liefert die Anzahl der eingelesenen Objekte
- der Zustand des Datenstroms kann mit `feof` und `ferror` untersucht werden

```
size_t fwrite(const void *ptr, size_t size, size_t nobj,
              FILE *file)
```

- schreibt `nobj` Objekte der Größe `size` aus dem Vektor `ptr` in den Datenstrom
- liefert die Anzahl der ausgegebenen Objekte

Direkte Ein- und Ausgabe (2)

Vorteile von Textdateien:

- mit Editor lesbar und änderbar
- plattformunabhängig

Nachteile von Textdateien:

- hoher Speicherplatzbedarf
- hohe Zugriffszeiten
- nur auf `char` und `char *` kann direkt zugegriffen werden, alle anderen Datentypen müssen konvertiert werden

Direkte Ein- und Ausgabe: Beispiel

```
int saveToFile(student_t *s, int n, char *filename) {
    int x;
    FILE *file;

    file = fopen(filename, "wb");
    if (file == NULL)
        return -1;

    x = fwrite(s, sizeof(student_t), n, file);
    if (x != n)
        perror(filename);
    return fclose(file);
}
```

Direkte Ein- und Ausgabe: Beispiel (2)

```
int readFromFile(student_t *s, int n, char *filename) {
    int x;
    FILE *file;

    file = fopen(filename, "rb");
    if (file == NULL)
        return -1;

    x = fread(s, sizeof(student_t), n, file);
    if (x != n)
        perror(filename);
    return fclose(file);
}
```


Direkte Ein- und Ausgabe: Beispiel (3)

```
int main(void) {
    student_t stud[5], *s;

    ...
    saveToFile(stud, 5, filename);

    s = (student_t *) malloc(5 * sizeof(student_t));
    readFromFile(s, 5, filename);
    for (i = 0; i < 5; i++)
        printf("Student: %s %s, %hd, %hd, %ld\n",
                s[i].name, s[i].vorname,
                s[i].alter, s[i].fb, s[i].matrikelnr);

    return 0;
}
```


Direkte Ein- und Ausgabe (3)

Achtung: Bei `fwrite` werden nur die Werte von Zeigern gespeichert, nicht der Inhalt, auf den der Zeiger zeigt!

```
typedef struct {
    char name[20];
    char vorname[20];
    short alter, fb;
    long matrikelnr;
} student_t;

void createStudent(student_t *s, char *name, char *vorn,
        short alter, short fb, long matrikelnr) {
    strncpy(s->name, name, 20);
    strncpy(s->vorname, vorn, 20);
    ...
}
```


Positionieren in Dateien

`int fseek(FILE *file, long offset, int origin)`

- Dateiposition für `file` setzen, nachfolgende Lese- oder Schreiboperation greift auf Daten ab dieser Position zu.
- neue Position ergibt sich aus Addition von `offset` Bytes zu `origin`
- mögliche Werte für `origin`: Dateianfang `SEEK_SET`, aktuelle Position `SEEK_CUR`, Dateiende `SEEK_END`
- liefert einen von Null verschiedenen Wert bei Fehler

`long ftell(FILE *file)`

- liefert die aktuelle Dateiposition oder `-1L` bei Fehler

Positionieren in Dateien (2)

```
void rewind(FILE *file)
```

- analog zu `fseek(file, 0L, SEEK_SET); clearerr(file);`

```
int fgetpos(FILE *file, fpos_t *ptr)
```

- speichert aktuelle Position für den Datenstrom bei `*ptr`
- liefert einen von Null verschiedenen Wert bei Fehler

```
int fsetpos(FILE *file, const fpos_t *ptr)
```

- positioniert `file` auf die Position, die von `fgetpos` in `*ptr` abgelegt wurde
- liefert einen von Null verschiedenen Wert bei Fehler

Positionieren in Dateien: Beispiel

```
#include <stdio.h>
void main(void) {
    int z;
    char line[10];          /* feste Satzlänge: 10 */
    FILE *file;          /* keine Fehlerbehandlung */

    file = fopen("_seek.txt", "r");

    printf("gehe zu Zeile ");
    scanf("%d", &z);
    fseek(file, (z-1) * 10, SEEK_SET);
    fgets(line, 10, file);
    printf("%s\n", line);
    fclose(file);
}
```


Tests für Zeichenklassen: `ctype.h`

Funktionen zum Testen von Zeichen. Jede Funktion

- hat ein `int`-Argument, dessen Wert entweder `EOF` ist oder als `unsigned char` dargestellt werden kann.
- hat den Rückgabetyp `int`.
- liefert einen Wert ungleich Null, wenn das Argument die beschriebene Bedingung erfüllt.

Zusätzlich: Funktionen zur Umwandlung zwischen Groß- und Kleinbuchstaben.

`tolower(c)`	Umwandlung in Kleinbuchstaben
`toupper(c)`	Umwandlung in Großbuchstaben

Tests für Zeichenklassen (2)

Funktion	Beschreibung
`isalnum(c)`	`isalpha(c)` oder `isdigit(c)` ist erfüllt
`isalpha(c)`	`isupper(c)` oder `islower(c)` ist erfüllt
`iscntrl(c)`	Steuerzeichen
`isdigit(c)`	dezimale Ziffer
`isgraph(c)`	sichtbares Zeichen, kein Leerzeichen
`islower(c)`	Kleinbuchstabe, kein Umlaut oder ß
`isprint(c)`	sichtbares Zeichen, auch Leerzeichen
`isspace(c)`	Leerzeichen, Seitenvorschub, ...
`isupper(c)`	Großbuchstabe, kein Umlaut oder ß
`isxdigit(c)`	hexadezimale Ziffer

Hilfsfunktionen: `stdlib.h`

Umwandlung von Zeichenketten in Zahlen:

- `double strtod(const char *str, char **endp)` und
- `long strtol(const char *str, char **endp, int base)`

wandeln den Anfang der Zeichenkette `str` in `double`/`long` um, dabei wird Zwischenraum am Anfang ignoriert.

- `strtoul` analog zu `strtol`, Resultattyp `unsigned long`

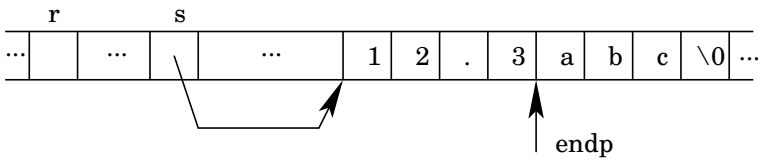
Die Funktionen speichern einen Zeiger auf den nicht umgewandelten Rest der Zeichenkette bei `*endp`.

Wert von `base`: $2, \ldots, 36$. Umwandlung erfolgt unter der Annahme, das die Eingabe in dieser Basis repräsentiert ist.

Erinnerung: call by reference

Warum ist `endp` als Zeiger auf Zeiger auf `char` definiert?

```
double d;
char *r, *s = "12.3abc";
```

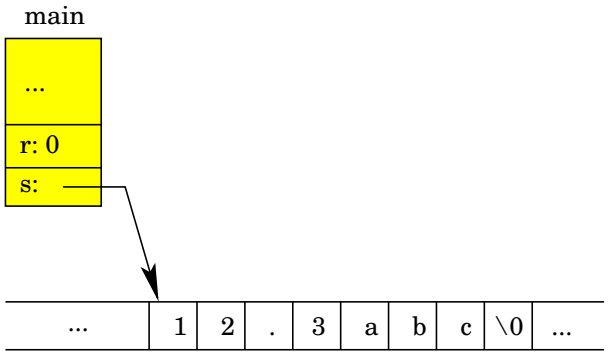


Damit der Wert von `r` in der Funktion `strtod` geändert werden kann, muss die Adresse von `r` übergeben werden.

```
d = strtod(s, &r);
```

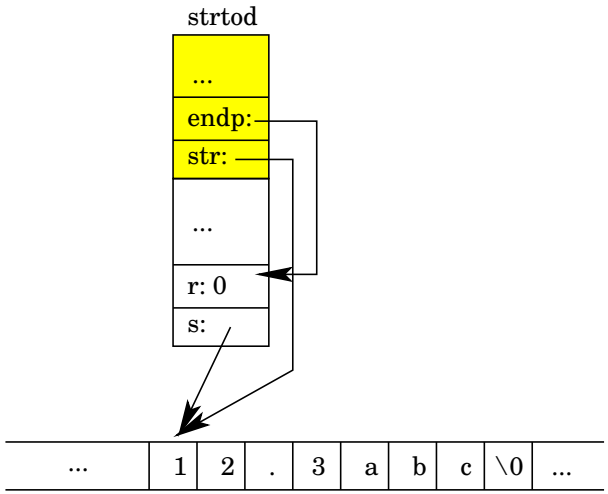

Erinnerung: call by reference (2)

Initial:



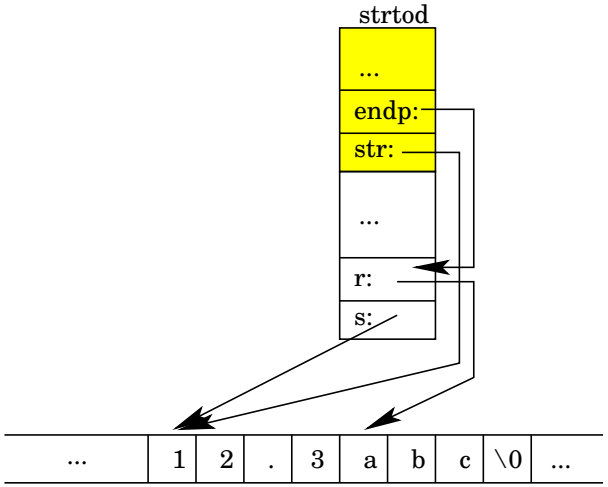
Erinnerung: call by reference (3)

Aufruf von `strtod`:



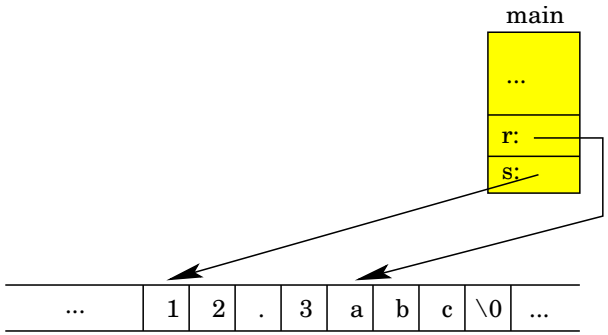
Erinnerung: call by reference (4)

vor dem Verlassen von `strtod`:



Erinnerung: call by reference (5)

Resultat:



Umwandlung von Zeichenketten: Beispiel

```
...
void main(int argc, char *argv[]) {
    int i;   char *r;   double d;   long l;

    for (i = 1; i < argc; i += 2) {
        if (strcmp(argv[i], "-d") == 0) {
            d = strtod(argv[i + 1], &r);
            printf("%d. Parameter: %f\n", i + 1, d);
        } else if (strcmp(argv[i], "-l") == 0) {
            l = strtol(argv[i + 1], &r, 0);
            printf("%d. Parameter: %ld\n", i + 1, l);
        }
        printf("Nicht umgewandelt: %s\n", r);
    }
}
```


Umwandlung von Zeichenketten: Beispiel (2)

Obiges Programm liefert bei dem Aufruf

```
strUmw -d 12.3abc -l 789.1abc -l 0X23AGH -l 022.2xy
```

die folgende Ausgabe:

```
2. Parameter: 12.300000
Nicht umgewandelt: abc
4. Parameter: 789
Nicht umgewandelt: .1abc
6. Parameter: 570
Nicht umgewandelt: GH
8. Parameter: 18
Nicht umgewandelt: .2xy
```

`base = 0` → übliche Interpretation für `int`-Konstanten

Einfache Umwandlung von Zeichenketten

`double atof(const char *s)`

- wandelt `s` in `double` um
- analog zu `strtod(s, (char **)NULL)`

`int atoi(const char *s)`

- wandelt `s` in `int` um
- analog zu `(int)strtol(s, (char **)NULL, 10)`

`int atol(const char *s)`

- wandelt `s` in `long` um
- analog zu `strtol(s, (char **)NULL, 10)`

Umwandlung von Zeichenketten: Beispiel

```
...
void main(int argc, char *argv[]) {
    int i, *arr;

    argc -= 1;
    arr = (int *) malloc(argc * sizeof(int));
    for (i = 0; i < argc; i++)
        arr[i] = atoi(argv[i + 1]);

    sort(arr, argc);

    for (i = 0; i < argc; i++)
        printf("%d\n", arr[i]);
}
```


Pseudo-Zufallszahlen

Die Kennzeichen einer Zufallsfolge sind:

- Zahlen entstammen einem gegebenen Zahlenbereich.
- Zahlen sind unabhängig voneinander: aus der Kenntnis der ersten `n` Zahlen kann nicht auf die `n+1`te Zahl geschlossen werden.
- Zahlen unterliegen gegebener Häufigkeitsverteilung.

Beispiel: Würfel

- Zahlen im Bereich 1 bis 6
- Gleichverteilung: Würfelt man genügend häufig, dann zeigt sich, dass die Zahlen 1 bis 6 ungefähr gleich oft erscheinen.

Pseudo-Zufallszahlen (2)

Einfache Zufallszahlen zwischen 0 und m:

- $x_0 \in [0 \ldots m-1]$ wird vorgegeben
- $x_{n+1} = (a \cdot x_n + 1) \bmod m$

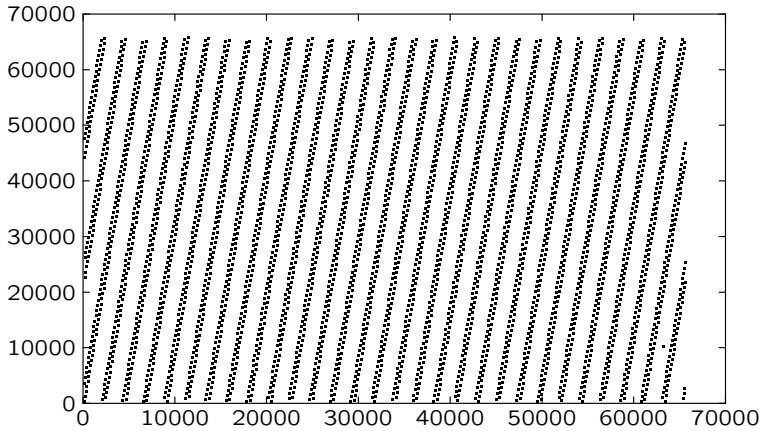
Beispiel: $x_{n+1} = (3423 \cdot x_n + 1) \bmod 2^{16}$

Die ersten 42 Werte für $x_0 = 0$:

0	1	3424	54945	53952	62785	20512
23521	34176	2689	29408	289	6208	16321
30112	50785	35584	38145	22624	43937	56768
2625	6944	45281	4224	40833	48608	54817
9024	21697	16544	7009	5632	10753	41824
32929	59584	8001	58912	1505	39808	13441

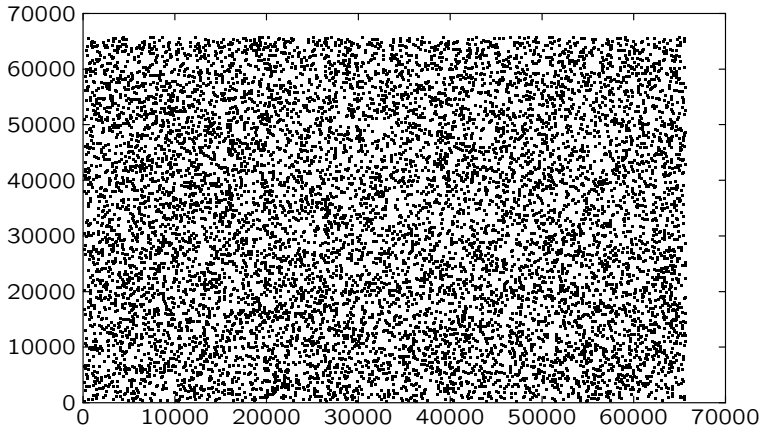
Pseudo-Zufallszahlen: Güte

Stelle je zwei Zufallszahlen als Koordinaten x, y dar. Bei *guter* Gleichverteilung ist das resultierende Bild gleich dicht mit Punkten gefüllt. → obige Zufallsfolge ist schlecht!



Pseudo-Zufallszahlen: Güte (2)

Gut: modifizierte Zufallsfolge $x_{n+1} = (3421 \cdot x_n + 1) \bmod 2^{16}$ mit $x_0 = 0$.



Pseudo-Zufallszahlen in C

`int rand(void)`

- liefert eine ganzzahlige Pseudo-Zufallszahl im Bereich von 0 bis `RAND_MAX` (mindestens $32767 = 2^{15} - 1$, ebenfalls definiert in `stdlib.h`)

`void srand(unsigned int seed)`

- benutzt `seed` als Ausgangswert für eine neue Folge von Pseudo-Zufallszahlen (entspricht x_0)

Die Güte des in C verwendeten Zufallszahlengenerators ist für *normale* Anwendungen ausreichend.

Pseudo-Zufallszahlen in C: Beispiel

```
...
#define N 10000

void main(void) {
    int i, arr[N];
    int seed = 1;

    srand(seed);
    for (i = 0; i < N; i += 2) {
        arr[i] = rand() % 200;
        arr[i + 1] = rand() % 200;
        printf("%3d, %3d\n", arr[i], arr[i + 1]);
    }
    printf("\n");
}
```

Funktionen für Zeichenketten: `string`

`char *strcpy(char *s, const char *ct)`
Zeichenkette `ct` in Vektor `s` kopieren, liefert `s`

`char *strncpy(char *s, const char *ct, int n)`
maximal `n` Zeichen aus `ct` in Vektor `s` kopieren, liefert `s`

`char *strcat(char *s, const char *ct)`
Zeichenkette `ct` an Zeichenkette `s` anhängen, liefert `s`

`char *strncat(char *s, const char *ct, int n)`
höchstens `n` Zeichen von `ct` an Zeichenkette `s` anhängen und mit `\0` abschließen, liefert `s`

Funktionen für Zeichenketten (2)

`int strcmp(const char *cs, const char *ct)`
Zeichenketten `cs` und `ct` vergleichen:

- liefert Wert < 0 wenn `cs < ct`,
- liefert 0 wenn `cs == ct` und
- liefert Wert > 0 wenn `cs > ct`.

`int strncmp(const char *cs, const char *ct, int n)`
höchstens `n` Zeichen von `cs` mit `ct` vergleichen

`char *strchr(const char *cs, int c)`
liefert Zeiger auf das erste `c` in `cs`, oder `NULL`

`char *strrchr(const char *cs, int c)`
liefert Zeiger auf das letzte `c` in `cs`, oder `NULL`

Funktionen für Zeichenketten (3)

`size_t strspn(const char *cs, const char *accept)`
liefert Anzahl der Zeichen am Anfang von `cs`, die sämtlich in `accept` vorkommen.

`size_t strcspn(const char *cs, const char *reject)`
liefert Anzahl der Zeichen am Anfang von `cs`, die sämtlich *nicht* in `reject` vorkommen.

`char *strpbrk(const char *cs, const char *ct)`
liefert Zeiger auf die Position in `cs`, an der irgendein Zeichen aus `ct` erstmals vorkommt, oder `NULL`.

`char *strstr(const char *cs, const char *ct)`
liefert Zeiger auf erste Kopie von `ct` in `cs`, oder `NULL`.

Funktionen für Zeichenketten (4)

`size_t strlen(const char *cs)`
liefert die Länge von `cs` (ohne `'\0'`)

`char *strerror(int n)`
liefert Zeiger auf Zeichenkette, die in der Implementierung für Fehler `n` definiert ist. Anwendung: `strerror(errno)`

Beispiel:

```
for (i = 0; i < 30; i++)
    printf("%s\n",
        strerror(i));
```

liefert:

```
Success
Operation not permitted
No such file or directory
No such process
Interrupted system call
...
```

Funktionen für Zeichenketten: Beispiel

`grep [-i] pattern file`
→ print lines matching [ignore-case] a pattern

```
...
char *uppercase(char **word) {
    int i, len;

    len = strlen(*word);
    for (i = 0; i < len; i++)
        (*word)[i] = toupper((*word)[i]);
    return *word;
}
```


Funktionen für Zeichenketten: Beispiel (2)

```
#define TRUE 1
#define FALSE 0
#define N 80

int main(int argc, char **argv) {
    FILE *file;
    int cnt, idx;
    char ignoreCase, pattern[N], *line;

    /* Aufruf formal korrekt? */
    if ((argc != 3) && (argc != 4)) {
        printf("try: %s [-i] pattern file\n", argv[0]);
        return 1;
    }
```


Funktionen für Zeichenketten: Beispiel (3)

```
    /* ignore-case? */
    if (strcmp(argv[1], "-i")) {
        ignoreCase = FALSE;
        idx = 2;
        strcpy(pattern, argv[1]);
    } else {
        ignoreCase = TRUE;
        idx = 3;
        strcpy(pattern, uppercase(&argv[2]));
    }

    file = fopen(argv[idx], "r");
    if (file == NULL)
        return 2;
```


Funktionen für Zeichenketten: Beispiel (4)

```
    /* Zeilenweise testen */
    cnt = 0;
    line = (char *) malloc(N * sizeof(char));
    while (fgets(line, N, file)) {
        cnt += 1;
        if (ignoreCase) {
            if (strstr(uppercase(&line), pattern))
                printf("%3d: %s", cnt, line);
        } else if (strstr(line, pattern))
            printf("%3d: %s", cnt, line);
    }

    fclose(file);
    return 0;
}
```

Funktionen für Zeichenketten (5)

`char *strtok(char *s, const char *ct)`
durchsucht `s` nach Zeichenfolgen, die durch Zeichen aus `ct` begrenzt sind.

Der erste Aufruf findet die erste Zeichenfolge in `s`, die nicht aus Zeichen in `ct` besteht. Die Folge wird abgeschlossen, indem das nächste Zeichen in `s` mit `\0` überschrieben wird. Resultat: Zeiger auf die Zeichenfolge.

Bei jedem weiteren Aufruf wird `NULL` anstelle von `s` übergeben. Solch ein Aufruf liefert die nächste Zeichenfolge, wobei unmittelbar nach dem Ende der vorhergehenden Suche begonnen wird.

Hinweis: Die Zeichenkette `ct` kann bei jedem Aufruf verschieden sein.

Funktionen für Zeichenketten: Beispiel

Abspeichern der Studentendaten als CSV-Datei (Comma Separated Values).

```
Huber,Hans Walter,42,3,12345678
Meier,Ulrike Maria,37,4,07654321
```

Einlesen der Studentendaten aus CSV-Datei:

```
char line[100];
...
liste = createList();
while (fgets(line, 100, file)) {
    s = (Student *)malloc(sizeof(Student));
    extractStudent(s, line);
    insert(liste, *s);
}
```


Funktionen für Zeichenketten: Beispiel (2)

Extrahieren der Studentendaten (Name, Vorname, Alter, FB, Matrikelnummer) mittels `strtok`.

Umwandeln der Strings (Alter, FB, Matrikelnummer) in Zahlen mittels `atoi` bzw. `atol`.

```
void extractStudent(student_t *student, char *data) {
   char *s, *name, *vorn;
   short alter, fb;
   long matrikelnr;

   /* extrahieren der Daten aus String 'data' */
   s = strtok(data, ";,:");
   name = (char *)malloc((strlen(s)+1) * sizeof(char));
   strcpy(name, s);
```


Funktionen für Zeichenketten: Beispiel (3)

```
   s = strtok(NULL, ";,:");
   vorn = (char *)malloc((strlen(s)+1) * sizeof(char));
   strcpy(vorn, s);
   alter = atoi(strtok(NULL, ";,:"));
   fb = atoi(strtok(NULL, ";,:"));
   matrikelnr = atol(strtok(NULL, ";,:"));

   /* zuweisen der Werte an 'student' */
   student->name = name;
   student->vorname = vorn;
   student->alter = alter;
   student->fb = fb;
   student->matrikelnr = matrikelnr;
}
```

Vollständiges Beispiel

Schneller Zugriff durch Index: erster Buchstabe des Namens
Voraussetzung: sortierte Datensätze, keine Umlaute

```
#define LENGTH 100
int len[26] = {0};
char line[LENGTH];

void main(int argc, char *argv[]) {
    char c, *filename = "_student.txt";

    createIndex(filename);
    printf("erster Buchstabe des Namens: ");
    scanf("%c", &c);
    printStudents(c, filename);
}
```


Vollständiges Beispiel (2)

```
int createIndex(char *filename) {
    FILE *f;

    f = fopen(filename, "r");
    if (f == NULL) {
        perror(filename);
        return -1;
    }
    while (fgets(line, LENGTH, f)) {
        char c = tolower(line[0]);
        len[c - 'a'] += strlen(line);
    }
    return fclose(f);
}
```


Vollständiges Beispiel (3)

```
int getIndex(char c) {
    int e, i, idx = 0;

    e = tolower(c) - 'a';
    for (i = 0; i < e; i++)
        idx += len[i];
    return idx;
}
```


Vollständiges Beispiel (4)

```
void printStudents(char c, char *filename) {
    FILE *f;
    int idx = getIndex(c);

    f = fopen(filename, "r");
    if (f == NULL) return;

    fseek(f, idx, SEEK_SET);   /* Fehlerbehandlung?? */
    while (fgets(line, LENGTH, f)
            && toupper(line[0]) == toupper(c))
        extractStudent(line);
    fclose(f);
}
```

Vollständiges Beispiel (5)

```
void extractStudent(char *data) {
    char name[20], vorname[20];
    short alter, fb;
    long matrikelnr;

    strcpy(name, strtok(data, ";,:"));
    strcpy(vorname, strtok(NULL, ";,:"));
    alter = atoi(strtok(NULL, ";,:"));
    fb = atoi(strtok(NULL, ";,:"));
    matrikelnr = atol(strtok(NULL, ";,:"));

    printf("%s, %s, %hd, %hd, %ld\n", name, vorname,
           alter, fb, matrikelnr);
}
```


Hilfsfunktionen: System

`void exit(int status)` beendet das Programm normal

- `atexit`-Funktionen werden in umgekehrter Reihenfolge ihrer Hinterlegung durchlaufen.
- Die Puffer offener Dateien werden geschrieben, offene Datenströme werden geschlossen.
- Die Kontrolle geht an die Umgebung des Programms zurück. 0 gilt als erfolgreiches Ende eines Programms.

`int atexit(void (*fcn)(void))` hinterlegt die Funktion `fcn`

- Liefert einen Wert ungleich 0, wenn die Funktion nicht hinterlegt werden konnte.

System: Beispiel

```
...
void exitFkt(void) {
    printf("Aufruf der atexit-Funktion!\n");
}

int main(int argc, char *argv[]) {
    int r;
    ...
    r = atexit(&exitFkt);
    if (r != 0) {
        printf("Exit-Funktion _NICHT_ hinterlegt!\n");
        exit(1);
    }
    ...
}
```


Hilfsfunktionen: System (2)

`int system(const char *s)`

- liefert die Zeichenkette `s` an die Umgebung
- die Umgebung führt die Anweisung `s` aus
- Resultat: Rückgabewert (Status) des auszuführenden Kommandos, im Fehlerfall -1. **Zur Erinnerung:** `main` kann einen Integer-Wert zurückliefern!

`char *getenv(const char *name)`

- liefert den Inhalt der Umgebungsvariablen `name`, im Fehlerfall `NULL` (falls keine Variable des Namens existiert)
- Details sind implementierungsabhängig

System: Beispiel

```
...
int main(int argc, char *argv[]) {
    int r;
    char *s;

    s = getenv("PATH");
    printf("%s\n", s);

    r = system("cp quelle ziel");
    if (r != 0) {
        printf("%d: cp _NICHT_ ausgeführt!\n", r);
        exit(1);
    }
    exit(0);
}
```


Mathematische Funktionen: `math.h`

In `math.h` vereinbarte Konstanten (Auszug):

`M_E`	2.7182818284590452354	e
`M_LOG2E`	1.4426950408889634074	$\log_2(e)$
`M_LOG10E`	0.43429448190325182765	$\log_{10}(e)$
`M_LN2`	0.69314718055994530942	$\log_e(2)$
`M_LN10`	2.30258509299404568402	$\log_e(10)$
`M_PI`	3.14159265358979323846	π
`M_PI_2`	1.57079632679489661923	$\pi/2$
`M_PI_4`	0.78539816339744830962	$\pi/4$
`M_1_PI`	0.31830988618379067154	$1/\pi$
`M_SQRT2`	1.41421356237309504880	$\sqrt{2}$
`M_SQRT1_2`	0.70710678118654752440	$1/\sqrt{2}$

Mathematische Funktionen (2)

Im weiteren gilt: `x,y: double` und `n: int`
Alle Funktionen liefern einen Wert vom Typ `double`.

`sin(x)`, `cos(x)`, `tan(x)`	Sinus, Cosinus, Tangens ...
`asin(x)`, `acos(x)`, `atan(x)`	... inverse Funktionen
`sinh(x)`, `cosh(x)`, `tanh(x)`	... hyperbolische Funktionen
`exp(x)`	e^x
`log(x)`	$\ln(x)$
`log10(x)`	$\log_{10}(x)$
`sqrt(x)`	$\sqrt{x}$ (Fehler: $x < 0$)
`pow(x, y)`	x^y (Fehler: $x < 0$ und $y \notin \mathbb{Z}$)
`fabs(x)`	Absolutwert (Betrag) von x
`ldexp(x, n)`	$x \cdot 2^n$

Mathematische Funktionen (3)

`ceil(x)` $\widehat{=} \lceil x \rceil$
kleinster ganzzahliger Wert, der nicht kleiner als x ist

`floor(x)` $\widehat{=} \lfloor x \rfloor$
größter ganzzahliger Wert, der nicht größer als x ist

`frexp(x, int *exp)`
zerlegt x in normalisierte Mantisse (Resultat) und in eine Potenz von 2 (wird in `*exp` abgelegt)

`modf(x, double *ip)`
zerlegt x in einen ganzzahligen Teil (wird bei `*ip` abgelegt) und in einen Rest (Resultat).

`fmod(x, y)` $\widehat{=} x - \lfloor \frac{x}{y} \rfloor \cdot y$

Mathematische Funktionen (4)

In `errno.h` befinden sich die Konstanten `EDOM` und `ERANGE`, die Fehler im Argument- und Resultatbereich der Funktionen anzeigen. **Wichtig: `errno` vor Funktionsaufruf löschen!**

```
errno = 0;
erg = sqrt(-2.0);
if (errno == EDOM)    /** Argumentfehler **/
    printf("-2.0: wrong argument for sqrt!\n");
else printf("sqrt(-2.0) = %f\n", erg);

errno = 0;
erg = exp(800);
if (errno == ERANGE)  /** Resultatfehler **/
    printf("exp(800): out of range!\n");
else printf("exp(800) = %f\n", erg);
```


Datum und Uhrzeit: `time.h`

`clock_t`, `time_t`: arithmetische Typen, repräsentieren Zeiten

`struct tm` enthält Komponenten einer Kalenderzeit:

`tm_sec`	Sekunden nach der vollen Minute
`tm_min`	Minuten nach der vollen Stunde
`tm_hour`	Stunden seit Mitternacht
`tm_mday`	Tage im Monat (1..31)
`tm_mon`	Monate **seit** Januar (0..11)
`tm_year`	Jahre **seit** 1900
`tm_wday`	Tage seit Sonntag (0..6)
`tm_yday`	Tage seit dem 1. Januar (0..365)
`tm_isdst`	Kennzeichen für Sommerzeit

Datum und Uhrzeit (2)

`time_t time(time_t *t)`
liefert die aktuelle Kalenderzeit, Resultat auch bei `*t`

`double difftime(time_t time2, time_t time1)`
liefert `time2 - time1` ausgedrückt in Sekunden

`struct tm *gmtime(const time_t *t)`
wandelt die Kalenderzeit `*t` in *Coordinated Universal Time* UTC (historisch: Greenwich Mean Time)

`struct tm *localtime(const time_t *t)`
wandelt die Kalenderzeit `*t` in Ortszeit

Datum und Uhrzeit (3)

```
#include <time.h>

void main() {
    time_t t;
    struct tm *d;

    t = time(0);          /* #Sekunden seit 1.1.1970 */
    d = localtime(&t);    /* Sekunden -> lokale Zeit */
    printf("Datum: %02d.%02d.%04d\n",
        d->tm_mday, d->tm_mon + 1, d->tm_year + 1900);
    printf("Zeit: %02d:%02d\n", d->tm_hour, d->tm_min);
    d = gmtime(&t);
    printf(" GMT: %02d:%02d\n", d->tm_hour, d->tm_min);
}
```

Datum und Uhrzeit (4)

```
size_t strftime(char *s, size_t smax, const char *fmt,
                const struct tm *t)
```

formatiert Datum und Zeit aus `*t` unter Kontrolle von `fmt` nach `s`, analog zu `sprintf`

`%a` `%A`	abgekürzter/voller Name des Wochentags
`%b` `%B`	abgekürzter/voller Name des Monats
`%c`	Datum und Uhrzeit
`%d`	Tag im Monat (1..31)
`%H`	Stunde (0..23)
`%j`	Tag im Jahr (1..366)
`%U`	Wochen im Jahr

Datum und Uhrzeit (5)

Zeitmessung im Programm:

```
...
int main(int argc, char **argv) {
    time_t time1, time2;
    double diff;
    ...
    time1 = time(0);
    /* zu messende Programmstelle */
    ...
    time2 = time(0);
    diff = difftime(time2, time1);
    ...
}
```


Datum und Uhrzeit (6)

Formatierung von Datum und Zeit:

```
void main() {
    char datum[80];
    time_t t;
    struct tm *d;

    t = time(0);
    d = localtime(&t);

    strftime(datum, 80, "%a, %d. %B %Y", d);
    printf("%s\n", datum);
}
```

liefert zum Beispiel: `Wed, 12. January 2005`

Datum und Uhrzeit (7)

Zeitmessung unter Unix/Linux: `gettimeofday`

```
#include <sys/time.h>
void main(int argc, char *argv[]) {
    long t;
    struct timeval t1, t2;
    ...
    gettimeofday(&t1, NULL);
    doSomething(args);
    gettimeofday(&t2, NULL);

    t  = t2.tv_sec * 1000000 + t2.tv_usec;
    t -= (t1.tv_sec * 1000000 + t1.tv_usec);
    printf("time: %f ms\n", ((float)t) / 1000);
}
```

Datum und Uhrzeit (8)

Zeitmessung unter Windows:

- `getSystemTime`
- `getSystemTimeAsFileTime`

Verzeichnisse: `dirent.h`

`DIR *opendir(const char *name)`
öffnet das angegebene Verzeichnis und liefert einen Datenstrom analog zu `fopen`.

`int closedir(DIR *dir)`
schließt den angegebenen Datenstrom analog zu `fclose`.

`struct dirent *readdir(DIR *dir)`
liefert einen Zeiger auf eine Struktur, die den nächsten Eintrag im Verzeichnis-Datenstrom repräsentiert.

`struct dirent`

`d_type`	Typ der Datei (Verzeichnis, Link, Datei, ...)
`d_name`	Name der Datei

Verzeichnisse: Beispiel (1)

Verzeichnis-Inhalt anzeigen: ohne Fehlerbehandlung!

```
void main(int argc, char *argv[]) {
   DIR *dir;
   struct dirent *entry;

   dir = opendir(argv[1]);
   while ((entry = readdir(dir)) != 0) {
      if (entry->d_type == 4)   /**  directory   **/
         printf("Verzeichnis: %s\n", entry->d_name);
      if (entry->d_type == 8)   /** regular file **/
         printf("Datei: %s\n", entry->d_name);
   }
   closedir(dir);
}
```


Verzeichnisse: Beispiel (2)

Verzeichnis- und Unterverzeichnisinhalt anzeigen:

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

int main(int argc, char **argv) {
    if (argc != 2) {
        printf("usage: %s path\n", argv[0]);
        return 1;
    }

    getDirEntries(argv[1]);
    return 0;
}
```

Verzeichnisse: Beispiel (3)

```
void getDirEntries(char *dirname) {
    DIR *dir;
    char subdir[100];
    struct dirent *entry;

    dir = opendir(dirname);
    if (dir == NULL) {
        printf("could not open %s\n", dirname);
        return;
    }
    ...
```


Verzeichnisse: Beispiel (4)

```
    while ((entry = readdir(dir)) != NULL) {
        if ((strcmp(entry->d_name, ".") == 0)
                || (strcmp(entry->d_name, "..") == 0))
            continue;
        if (entry->d_type == 4) {  /** directory **/
            strcpy(subdir, dirname);
            strcat(subdir, "/");
            strcat(subdir, entry->d_name);
            getDirEntries(subdir);
        }
        if (entry->d_type == 8)   /** regular file **/
            printf("%s\n", entry->d_name);
    }
    closedir(dir);
}
```


Erweiterungen im C99 Standard

inline functions: Compiler-Hinweis, jeder Aufruf der Funktion ist durch Einfügen des Codes der Anweisungen des Funktionsrumpfs zu ersetzen.

Beispiel:

```
inline int max(int i, int j) {
    return (i > j) ? i : j;
}
```

⇒ spart das Erzeugen von Variablen sowie das Kopieren von Werten für Argumente und Funktionswert!

⇒ nur sinnvoll für Funktionen, deren Rumpf nur wenige Anweisungen enthalten

Erweiterungen im C99 Standard (2)

Variablen-Deklaration sind nun an vielen Stellen im Code erlaubt, nicht nur zu Beginn eines Blocks

Beispiel:

```
#include <stdio.h>
main() {
    for (int i = 0; i < 10; i++)
        printf("%2d: Hello, world!\n", i);

    int x = 1;
    while (x < 10)
        printf("x = %3d\n", x++);
}
```

⇒ kein sinnvolles Feature, Code wird schlechter lesbar

Erweiterungen im C99 Standard (3)

neue Datentypen:

- `long long int` schließt Lücke zwischen 32- und 64-Bit
- `complex` Darstellung komplexer Zahlen + Arithmetik

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

main() {
   complex c = 1.0f + 1.0f * _Complex_I;

   printf("c+c = %f+i%f\n", creal(c+c), cimag(c+c));
   printf("c-c = %f+i%f\n", creal(c-c), cimag(c-c));
   printf("c*c = %f+i%f\n", creal(c*c), cimag(c*c));
   printf("c/c = %f+i%f\n", creal(c/c), cimag(c/c));
}
```


Erweiterungen im C99 Standard (4)

erweiterte/neue Bibliotheken

- `stdint.h` Definition von `long long int`
- `complex.h` Darstellung komplexer Zahlen + Arithmetik
- `snprintf(char *str, size_t size, char *format, ...)` schreibt maximal `size` Zeichen nach `str`, Formatierung erfolgt anhand `format`
- `va_copy(va_list dest, va_list src)`

Erweiterungen im C99 Standard (5)

heute üblich: **Unicode**-Zeichensatz

- C90 definert **breite Zeichen**: Datentyp `wchar_t`
- C99 definiert alle Bibliotheksfunktionen über Zeichenketten auch in einer Version für breite Zeichen

```
#include <wchar.h>
int main(void) {
    wchar_t ustr[20] = L"abcde";
    int len;

    swprintf(ustr, L"%4i", 1234);
    len = wcslen(ustr);
    return 0;
}
```


Erweiterungen im C99 Standard (6)

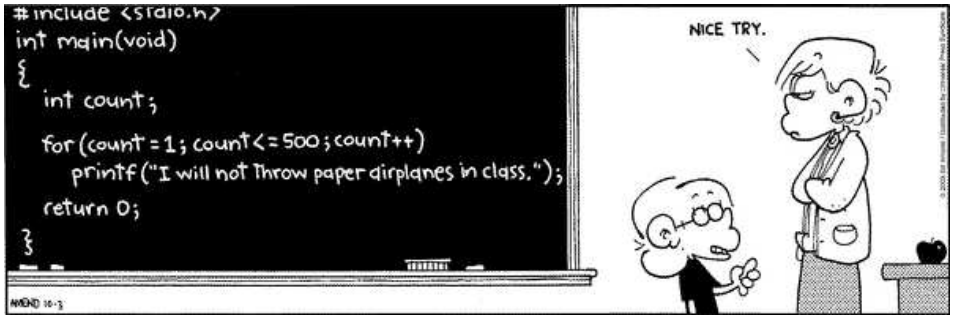
- **einzeilige Kommentare wie in C++:**

```
// Hauptprogramm
int main(int argc, char *argv[]) {
    int upper;  // obere Grenze
    int lower;  // untere Grenze
    ...
```

- **Arrays variabler Länge:**

```
int len = atoi(argv[1]);
int arr[len];    // nach ISO-C90 verboten
```

Was noch bleibt: Anwenden



Was noch bleibt: Programmentwicklung

Kenntnis der Programmiersprache: notwendige, aber nicht ausreichende Bedingung, um gute Software zu schreiben.

Software-Engineering:

- Ende der 60er Jahre: Software-Krise, Komplexität der Programme wuchs den Entwicklern über den Kopf, Fehlerbeseitigung führt zu neuen Fehlern.
- Erkenntnis: Software ist Ergebnis von Ingenieurtätigkeit
- Software ist industrielles Produkt mit definierten Qualitätsmerkmalen, das methodischer Planung, Entwicklung, Herstellung und Wartung bedarf.
- Es werden Methoden und Werkzeuge benötigt. (CASE-Tool: Computer Aided Software Engineering)

Programmentwicklung: Phasenmodell

Hier nur eine kurze Übersicht, näheres in der Vorlesung *Software-Engineering* bei Prof. Dr. Beims.

Lebenszyklus von Software: (software life cycle)

Problemanalyse: Zusammen mit dem Auftraggeber wird das Problem definiert und analysiert. Dient der genauen Spezifikation des Produkts.

Festlegen der Funktionalität, untersuchen der Durchführbarkeit, prüfen der ökonomischen Sinnhaftigkeit.

Resultat: Pflichtenheft, Benutzerhandbuch, Projekt- und Testplan.

Programmentwicklung: Phasenmodell (2)

Entwurfsphase: das Software-System wird als Ganzes entworfen und in Teile zerlegt (→ **Grobentwurf**), mit den Teilen ebenso verfahren (→ **Feinentwurf**).

Zerlegen in Teilprobleme, bis Aufgabe überschaubar:

- Das Programm ist die Gesamtheit der Module.
- Die Schnittstellen zwischen den Modulen müssen exakt definiert sein.

Auch unter Zeitdruck sind diese Tätigkeiten durchzuführen: **Je früher man mit dem Codieren beginnt, desto später ist man damit fertig.**

Denn: der Überblick geht verloren, die Aufgabe erscheint schwierig, Lösungen sind unnötig kompliziert.

Programmentwicklung: Phasenmodell (3)

Zerlegen in überschaubare Einheiten führt oft auf bereits gelöste Probleme (Top-Down-Methode). Die Analyse steht im Vordergrund. Geeignet für Entwicklung neuer Produkte.

Bottom-Up-Methode: Zusammenstellen von fertigen Bausteinen zu einem Produkt. Ungeeignet für Entwicklungsphase: verlieren in Details, Ziel wird aus dem Auge verloren. Geeignet für die Implementierungsphase.

Modularisierung:

- Eine Funktion führt eine abgeschlossene Aufgabe aus.
- Quellcode einer Funktion: nicht länger als zwei Seiten.
- Funktionen innerhalb einer Datei: wenn sie gemeinsame Daten verwenden oder logisch zusammengehören.

Programmentwicklung: Phasenmodell (4)

Implementieren: Editieren, Übersetzen, Binden

Testen: Es ist in der Regel nicht möglich, zu beweisen, dass ein Programm für alle möglichen Eingaben korrekt ist. **Testen des Programms zeigt nur die Anwesenheit von Fehlern, nicht deren Abwesenheit.**

Um sicherzustellen, dass die Funktionalität des Programms auch nach Änderungen/Erweiterungen gegeben ist, müssen Testdatensätze gesammelt und Tests geschrieben werden. **Tests müssen vollständig automatisiert sein und ihre Ergebnisse selbst überprüfen**, ansonsten testet niemand.

Programmentwicklung: Phasenmodell (5)

Wartungs- und Pflegephase: Programmierer sind 80% der Zeit mit Wartung und Pflege von Software beschäftigt.
⇒ keine Tricks anwenden, die andere nicht verstehen oder nachvollziehen können

Konstruktive Voraussicht: Mangelhafter Entwurf führt oft in Sackgassen, da sich neue Aspekte nicht realisieren lassen → weitsichtiger Entwurf und Implementierung

- trotz aller Voraussicht sind nicht alle Erweiterungen vorhersehbar
- oft werden erwartete Erweiterungen nicht realisiert, so dass der Entwurf unnötig komplex ist

⇒ **Refactoring**

Programmentwicklung: Was noch fehlt

Qualitätssicherung: Maßnahmen, deren Ziel die Verbesserung der Qualität des Software-Produktes ist.

Problem: Wie definiert man Qualität? Länge von Prozeduren, Schachtelungstiefe von Bedingungen und Schleifen, Struktur arithmetischer Ausdrücke, ... ???

Alle Herstellungsphasen des Produkts sind betroffen.

Programmentwicklung: Was noch fehlt (2)

Dokumentation: Jede Phase des Software-Projekts muss dokumentiert sein.

Problem: Dokumentation muss aktuell sein, sonst ist sie wertlos.

- Spezielle Kommentarformate im Source-Code können zur PDF- oder HTML-Dokumentation verarbeitet werden → javadoc, doxygen
- Programmstruktur ist mittels UML-, Modul- und Struktogrammen darstellbar
- die Datenbankstruktur kann als Entity-Relationship Diagramm beschrieben werden

Programmentwicklung: Erfolge

Methoden, die erfolgreich in der Programmentwicklung eingesetzt werden:

- Strukturierte Programmierung
- Top-Down-Entwurf
- Modulare Programmierung
- Objektorientierte Programmierung

Algorithmen & Datenstrukturen

Einführung

Was ist ein Algorithmus?

- Verfahren zur Lösung eines Problems, unabhängig von Implementierung in konkreter Programmiersprache.
- Vorteil: Konzentrieren auf das Problem, nicht auf die Eigenarten der Sprache.

Gemeinsame Konzepte der Programmiersprachen:

- Wertzuweisungen/Ausdrücke (in C: `y = 2*x + c`)
- Bedingte Anweisung (in C: `if/else`, `switch/case`)
- Iterative Anweisung (in C: `for`, `while`, `do/while`)
- Ein- und Ausgabeanweisung (in C: `scanf`, `printf`)
- Prozedur-/Funktionsaufrufe

Einführung (2)

Literatur:

- J. Loeckx, K. Mehlhorn, R. Wilhelm: Grundlagen der Programmiersprachen. B.G. Teubner Verlag.
- K.C. Louden: Programming Languages. PWS Publishing Company.
- C. Ghezzie, M. Jazayeri: Konzepte der Programmiersprachen. Oldenbourg Verlag.
- H.-P. Gumm, M. Sommer: Einführung in die Informatik. Oldenbourg Verlag.

Einführung (3)

Welche Eigenschaften interessieren uns?

- **Korrektheit** (E. Dijkstra: Testen kann die Anwesenheit, aber nicht die Abwesenheit von Fehlern zeigen.)
- **Laufzeit** (wie schnell wird das Problem gelöst?)
- **Speicherplatz** (wird zusätzlicher Speicher benötigt?)

Weitere interessante Eigenschaften:

- **Kommunikationszeit** (parallele/verteilte Algorithmen)
- **Güte** (exakte oder approximative Lösung?)

Einführung (4)

Wichtiger als Performanz?

- Wartbarkeit/Erweiterbarkeit
- Entwicklungszeit/Einfachheit
- Zuverlässigkeit/Ausfallsicherheit
- Bedienbarkeit

$\Rightarrow$ *Software-Engineering* (Prof. Dr. Beims)

Einführung (5)

Wenn ein Problem mit verschiedenen Algorithmen gelöst werden kann:

- Wie bewerten/vergleichen wir Algorithmen?
- Wann ist ein Algorithmus besser als ein anderer?
- Was sind gute/schlechte Algorithmen?

$\Rightarrow$ Laufzeit messen und vergleichen **???**

Probleme beim Messen und Vergleichen der Laufzeit:

- unterschiedlich schnelle Hardware/gute Compiler
- unterschiedliche Betriebssysteme/Laufzeitumgebungen
- unterschiedliche Eingabedarstellungen/Datenstrukturen

Einführung (6)

Systemumgebung 1:

Hardware:	Pentium III mobile, 1GHz, 256 MB
Linux:	Kernel 2.4.10, gcc 2.95.3
Windows:	XP Home (SP1), Borland C++ 5.02

Systemumgebung 2:

Hardware:	Pentium M (Centrino), 1,5GHz, 512 MB
Linux:	Kernel 2.6.4, gcc 3.3.3
Windows:	XP Home (SP2), Borland C++ 5.5.1

Einführung (7)

Messergebnisse: Zahlen sortieren

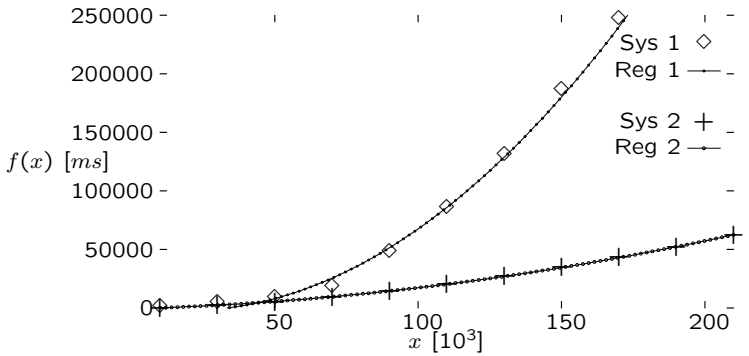
input size	System 1		System 2	
	Linux	XP [s]	Linux	XP [s]
8192	1	0	1	0
16384	3	2	2	1
32768	9	4	6	3
65536	34	17	21	9
131072	221	137	72	29

Problem: Bewertung der Messergebnisse

- Unterschiede aufgrund Compiler oder Betriebssystem?
- Skalierung: doppelte Eingabegröße, vierfache Laufzeit?

Einführung (8)

Laufzeit beschreibbar durch Polynom $ax^2 + bx + c$ **???**



Reg.1:	$10.6 \cdot x^2$	-	$399.4 \cdot x$	+	1507.3
Reg.2:	$1.1 \cdot x^2$	+	$69.9 \cdot x$	-	648.1

Einführung (9)

Minimieren der Summe der Fehlerquadrate:

$$F(x) := \sum_i (y_i - (ax_i^2 + bx_i + c))^2 \longrightarrow \min$$

Koeffizienten bestimmen durch quadratische Regression:

$$\frac{\partial F}{\partial a} = 0 \quad \frac{\partial F}{\partial b} = 0 \quad \frac{\partial F}{\partial c} = 0$$

$\Rightarrow$ es ist ein Gleichungssystem mit drei Gleichungen und drei Unbekannten zu lösen

Problem: Koeffizienten sind abhängig von Hard-/Software

Lösung: Korrekturfaktoren

Einführung (10)

Messen der Laufzeit:

- Implementieren in einer konkreten Sprache/Compiler.
- Festgelegt bei Ausführung: Rechner/Eingabemenge.

Probleme:

- Festlegen auf Norm nur schwer möglich.
- Ergebnisse lassen sich nur schwer übertragen.
- Aussage über Skalierung basiert auf Vermutung.
- Speicherbegrenzung: Paging/Swaping, Cache-Effekte

⇒ **Messen und Vergleichen der Laufzeiten ist nicht praktikabel oder sinnvoll!**

Bewertung von Algorithmen

Auswege:

- **idealisiertes Model** (RAM: Random Access Machine)
  - * Festgelegter Befehlssatz (Assembler-ähnlich)
  - * abzählbar unendlich viele Speicherzellen
  - ⇒ Laufzeit: Anzahl ausgeführter RAM-Befehle
  - ⇒ Speicherbedarf: Anzahl benötigter Speicherzellen
- **charakteristische Parameter ermitteln**
  - * Sortieren: Schlüsselvergleiche, Vertauschungen
  - * Arithmetik: Additionen, Multiplikationen

Mehr zu RAM, Turing-Maschinen und Komplexitätstheorie in der Vorlesung *Theoretische Informatik* (Prof. Dr. Dalitz)

Bewertung von Algorithmen (2)

Laufzeit und Speicherbedarf sind in der Regel abhängig von der Größe der Eingabe.

Warum ist die Komplexität der Algorithmen interessant?

Beispiel: Traveling Salesperson Problem

Gegeben: Eine Menge von Orten, die untereinander durch Wege verbunden sind. Die Wege sind unterschiedlich lang.
Gesucht: Eine Rundreise, die durch alle Orte genau einmal führt und unter allen Rundreisen minimale Länge hat. (Tourenplanung)

Bester bekannter Algorithmus: Laufzeit $\approx 2^n$ bei n Orten.

Bewertung von Algorithmen (3)

Annahme: Rechengeschwindigkeit beträgt 1GOp/s:

- 1.000.000.000 Schritte pro Sekunde
- 3.600.000.000.000 Schritte pro Stunde
- 86.400.000.000.000 Schritte pro Tag

⇒ Lösbare Problemgröße

- am Tag: 46 Städte
- im Jahr: 55 Städte
- in 100 Jahren: 61 Städte

in Deutschland: ≈ 5000 Städte

Bewertung von Algorithmen (4)

Frage: Löst ein schnellerer Rechner das Problem???
Antwort: **Nein!**

Projektion: Geschwindigkeit verdoppelte sich alle 1,5 Jahre.

- in 10 Jahren: Lösbare Problemgröße am Tag: 52 Orte
- in 100 Jahren: Lösbare Problemgröße am Tag: 112 Orte

Computer	Dauer	Anzahl Schritte
A	1 Tag	2^n
B	1 Tag	$2 \cdot 2^n = 2^{n+1}$

Nur theoretische Lösung, denn: Die Touren müssen jetzt geplant werden, nicht in 100 Jahren.

Bewertung von Algorithmen (5)

Lösung 1: bessere Algorithmen. **Speed is fun!**

Laufzeit	Dauer für 5000 Städte	#Städte pro Tag
$\approx n^2$	1 Sekunde	9.295.160
$\approx n^3$	2 Minuten	44.208
$\approx n^4$	7 Tage	3.048
$\approx n^5$	99 Jahre	612

Frage: Gibt es bessere Algorithmen für das Problem???

Bewertung von Algorithmen (6)

Lösung 2: parallele/verteilte Systeme
Problem auf mehreren Prozessoren/Computern gleichzeitig bearbeiten
$\Rightarrow$ *Parallele/Verteilte Systeme* (Prof. Dr. Ueberholz)

Lösbare Problemgröße am Tag bei linearem Speedup

#Rechner	TSP $\approx 2^n$	Matrixmultiplikation $\approx n^3$
1	46	44.208
100	53	205.197
1.000	56	442.083
10.000	59	952.440

Asymptotische Komplexität

Genaue Angabe der Komplexitätsfunktion ist oft schwierig oder unmöglich.

Unscharfe Aussagen: Abstrahieren von

- additiven Konstanten,
- konstanten Faktoren und
- Termen niedrigerer Ordnung.

Beispiel:

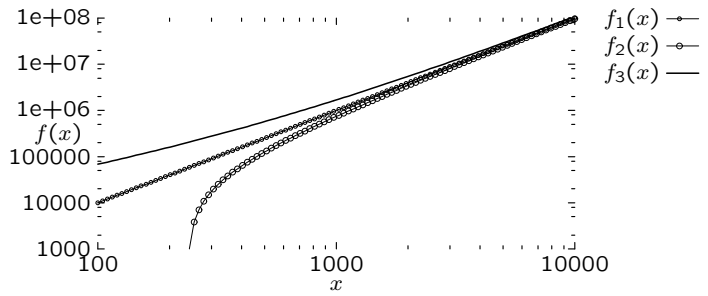
$$\begin{aligned} & 7 \cdot \log_2(n) + 5 \cdot n^3 + 3 \cdot n^4 \\ \leq\ & 7 \cdot n^4 + 5 \cdot n^4 + 3 \cdot n^4 \\ \leq\ & 15 \cdot n^4 \\ \approx\ & n^4 \end{aligned}$$

Asymptotische Komplexität (2)

Dürfen Terme niedriger Ordnung vernachlässigt werden???

Betrachten wir dazu die folgenden Funktionen:

$$\begin{aligned} f_1(x) &= x^2 \\ f_2(x) &= x^2 - 17 \cdot x \cdot \log_2(x) - 139 \cdot x - 1378 \\ f_3(x) &= x^2 + 64 \cdot x \cdot \log_2(x) + 241 \cdot x + 4711 \end{aligned}$$



Asymptotische Komplexität (3)

Anmerkungen:

- Bei kleinen Eingabegrößen sind die konstanten Faktoren und Terme niedriger Ordnung entscheidend.
- Große Konstanten: theoretisch gute Lösungen sind oft praktisch nicht akzeptabel.
- **Hard-/Software-abhängige Konstanten fallen nicht mehr ins Gewicht.**

Aufwandsklassen

Definition: Sei $g : \mathbb{N}_0 \to \mathbb{N}_0$ eine Funktion. Dann bezeichnet $O(g)$ die Menge der Funktionen, die asymptotisch höchstens so stark wachsen wie g. $\to g$ ist obere Schranke!

$$O(g) = \{f : \mathbb{N}_0 \to \mathbb{N}_0 \mid \exists c \in \mathbb{N} : \lim_{n\to\infty} \frac{f(n)}{g(n)} \leq c\}$$

Wir schreiben:	$O(n)$	für	$O(g)$	falls $g(n) = n$
	$O(n^k)$	für	$O(g)$	falls $g(n) = n^k$
	$O(\log(n))$	für	$O(g)$	falls $g(n) = \log(n)$
	$O(\sqrt{n})$	für	$O(g)$	falls $g(n) = \sqrt{n}$
	$O(2^n)$	für	$O(g)$	falls $g(n) = 2^n$

Aufwandsklassen (2)

Beispiele:

- $2 \cdot n^2 + 37 \cdot n^3 \in O(n^3)$
- $2 \cdot n^2 + 37 \cdot n^3 \in O(n^4)$
- $2 \cdot n^2 + 37 \cdot n^3 \notin O(n^2)$
- $42 \cdot n \cdot \log(n) + 3 \cdot n^2 \in O(n^2)$
- $17 \cdot \sqrt{n} + 139 \cdot n \in O(n)$
- $17 \cdot \sqrt{n} + 139 \cdot n \notin O(\log(n))$
- $928 \cdot n^4 + 0.7 \cdot 2^n \in O(2^n)$
- $c_1 \cdot n + c_2 \cdot n^2 + \ldots + c_k \cdot n^k \in O(n^k)$ für $c_1, \ldots, c_k$ konstant

Aufwandsklassen (3)

Definition: Sei $g : \mathbb{N}_0 \to \mathbb{N}_0$ eine Funktion. Dann bezeichnet $\Omega(g)$ die Menge der Funktionen, die asymptotisch mindestens so stark wachsen wie g. $\to g$ ist untere Schranke!

$$\Omega(g) = \{f : \mathbb{N}_0 \to \mathbb{N}_0 \mid \exists c \in \mathbb{N} : \lim_{n\to\infty} \frac{g(n)}{f(n)} \leq c\}$$

Beispiele:

- $2 \cdot n^2 + 37 \cdot n^3 \in \Omega(n^3)$
- $2 \cdot n^2 + 37 \cdot n^3 \in \Omega(n^2)$
- $2 \cdot n^2 + 37 \cdot n^3 \notin \Omega(n^4)$
- $42 \cdot n \cdot \log(n) + 3 \cdot n^2 \in \Omega(n)$
- $17 \cdot \sqrt{n} + 139 \cdot n \in \Omega(n)$
- $c_1 \cdot n + c_2 \cdot n^2 + \ldots + c_k \cdot n^k \in \Omega(n^k)$ für $c_1, \ldots, c_k$ konstant

Aufwandsklassen (4)

Definition: Sei $g : \mathbb{N}_0 \to \mathbb{N}_0$ eine Funktion. Dann bezeichnet $\Theta(g)$ die Menge der Funktionen, die asymptotisch genauso stark wie g wachsen.

$$\Theta(g) = \{f : \mathbb{N}_0 \to \mathbb{N}_0 \mid f \in O(g) \wedge f \in \Omega(g)\}$$

Beispiele:

- $2 \cdot n^2 + 37 \cdot n^3 \in \Theta(n^3)$
- $42 \cdot n \cdot \log(n) + 3 \cdot n^2 \in \Theta(n^2)$
- $17 \cdot \sqrt{n} + 139 \cdot n \in \Theta(n)$
- $928 \cdot n^4 + 0.7 \cdot 2^n \in \Theta(2^n)$
- $c_1 \cdot n + c_2 \cdot n^2 + \ldots + c_k \cdot n^k \in \Theta(n^k)$ für $c_1, \ldots, c_k$ konstant

Aufwandsklassen (5)

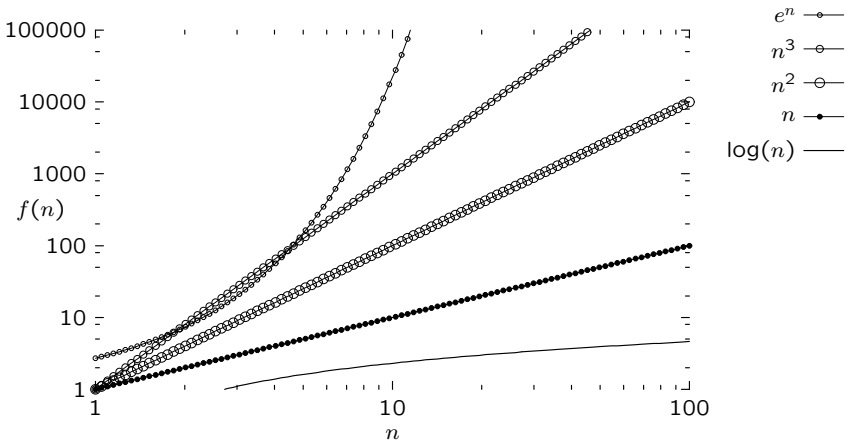
Wichtige Aufwandsklassen:

$O(1)$	konstant	$O(n^2)$	quadratisch
$O(\log(n))$	logarithmisch	$O(n^3)$	kubisch
$O(\log^k(n))$	poly-logarithmisch	$O(n^k)$	polynomiell
$O(n)$	linear	$O(2^n)$	exponentiell
$O(n \cdot \log(n))$			

Inklusionen der wichtigsten Aufwandsklassen:

$$O(1) \subset O(\log(n)) \subset O(\log^2(n)) \subset O(\sqrt{n}) \subset O(n) \subset O(n \cdot \log(n)) \subset O(n^2) \subset O(n^3) \subset O(2^n)$$

Vergleich der Aufwandsklassen



Vergleich der Aufwandsklassen (2)

GOp/sec	n^2	n^3	n^4	e^n
1	31.622	1.000	177	20
10	100.000	2.154	316	23
100	316.227	4.641	562	25
1000	1.000.000	10.000	1.000	27
$\Delta =$ $\cdot 10$	$\cdot 3.16$	$\cdot 2.15$	$\cdot 1.78$	$+2.3$

$O(n^2)$: $n_1 = \sqrt{1e9} = 31.622$

$n_2 = \sqrt{1e10} = \sqrt{1e9} \cdot \sqrt{10} = n_1 \cdot \sqrt{10}$

$O(e^n)$: $n_1 = \ln(1e9) = 20.723\ldots$

$n_2 = \ln(1e10) = \ln(1e9) + \ln(10) = n_1 + \ln(10)$

Vergleich der Aufwandsklassen (3)

Annahme: 1.000.000.000 Schritte pro Sekunde.

Lösbare Problemgröße bei verschiedenen Zeitvorgaben:

Aufwand	1 Sek	1 Min	1 Std	1 Tag
n^2	31.622	244.948	1.897.366	9.295.160
n^3	1.000	3.914	15.326	44.208
n^4	177	494	1.377	3.048
n^5	63	143	324	612
e^n	20	24	28	32

Komplexitätsmaße

Man unterscheidet die Laufzeit

- im besten Fall (best case)
- im Mittel (average case)
- im schlechtesten Fall (worst case)

Vergleich: Lineare Suche vs. binäre Suche

Algorithmus	best case	average case	worst case
lineare Suche	1	$N/2$	N
binäre Suche	1	$\log_2(N)$	$\log_2(N)$

Komplexitätsmaße (2)

Probleme bei average case:

- Worüber bildet man den Durchschnitt?
- Sind alle Eingaben der Länge N gleich wahrscheinlich (Gleichverteilung)?
- Technisch oft sehr viel schwieriger durchzuführen als worst-case Analyse.

Murphys Gesetz: Alles was schiefgehen kann, wird auch schiefgehen. $\Rightarrow$ Immer wenn ich das Programm ausführe, warte ich ewig.

Ungeeignet für kritische Anwendungen, bei denen maximale Reaktionszeiten garantiert werden müssen $\rightarrow$ *Echtzeitsysteme* (Prof. Dr. Quade)

Worst-Case Komplexität

Definition: (Worst-Case Komplexität)
W_n: Menge der zulässigen Eingaben der Länge n.
$A(w)$: Anzahl Schritte von Algorithmus A für Eingabe w.

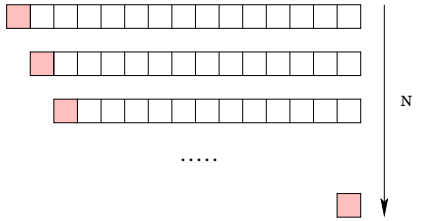
Worst-Case Komplexität (im schlechtesten Fall):

$$T_A(n) = \sup\{A(w) \mid w \in W_n\}$$

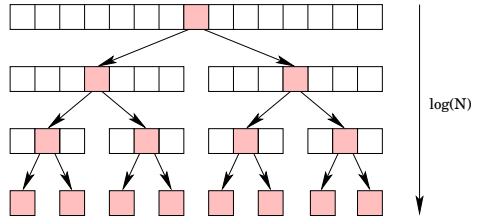
ist eine obere Schranke für die maximale Anzahl der Schritte, die Algorithmus A benötigt, um Eingaben der Größe n zu bearbeiten.

Worst-Case Komplexität: Beispiel

lineare Suche:



binäre Suche:



Zum Vergleich:

N	$\log(N)$
1.000.000	20
1.000.000.000	30
1.000.000.000.000	40

Average-Case Komplexität

Definition: (Average-Case Komplexität)
W_n: Menge der zulässigen Eingaben der Länge n.
$A(w)$: Anzahl Schritte von Algorithmus A für Eingabe w.

Average-Case Komplexität (erwarteter Aufwand):

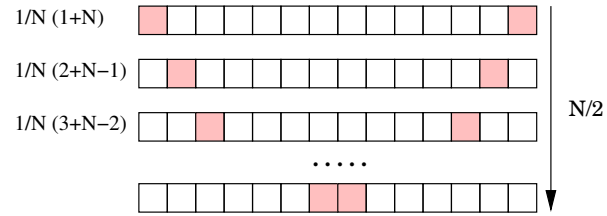
$$\overline{T}_A(n) = \frac{1}{|W_n|} \cdot \sum_{w \in W_n} A(w)$$

ist die mittlere Anzahl von Schritten, die Algorithmus A benötigt, um eine Eingabe der Größe n zu bearbeiten. Wir setzen hier eine Gleichverteilung voraus $\rightarrow$ arithmetischer Mittelwert

Average-Case Komplexität: Beispiel

lineare Suche:

Kosten: $1, \ldots, N$ Vergleiche
erwartete Kosten: $\frac{1}{N} \cdot (1 + 2 + 3 + \ldots + N)$

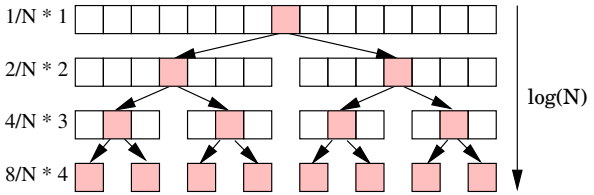


$$\frac{1}{N} \cdot (1 + 2 + 3 + \ldots + N) = \frac{1}{N} \cdot \frac{N(N+1)}{2} = \frac{N+1}{2}$$

Average-Case Komplexität: Beispiel (2)

binäre Suche:

Kosten:	$1, \ldots, \log(N)$
zur Vereinfachung:	$N = 2^x - 1$
erwartete Kosten:	$\frac{1}{N} \cdot (1 + 2 \cdot 2 + 4 \cdot 3 + \ldots + 2^{x-1} \cdot x)$



Average-Case Komplexität: Beispiel (3)

Behauptung:

$$\sum_{i=1}^{x} i \cdot 2^{i-1} = (x-1) \cdot 2^x + 1$$

Beweis mittels vollständiger Induktion:

I.A. $x = 1$:

$$1 \cdot 2^0 = 1 \cdot 1 = 1 \stackrel{!}{=} 0 \cdot 2^1 + 1 = 1$$

I.S. $x \to x + 1$:

$$\begin{aligned} \sum_{i=1}^{x+1} i \cdot 2^{i-1} &= \sum_{i=1}^{x} i \cdot 2^{i-1} + (x+1) \cdot 2^x \\ &\stackrel{I.V.}{=} (x-1) \cdot 2^x + 1 + (x+1) \cdot 2^x \\ &= 2x \cdot 2^x + 1 = x \cdot 2^{x+1} + 1 \end{aligned}$$

Average-Case Komplexität: Beispiel (4)

Aus der Annahme $N = 2^x - 1$ folgt: $\log_2(N+1) = x$

Somit ergibt sich:

$$\begin{aligned} \frac{1}{N} \cdot \sum_{i=1}^{x} i \cdot 2^{i-1} &= \frac{1}{N} \cdot [(x-1) \cdot 2^x + 1] \\ &= \frac{1}{N} \cdot [(\log_2(N+1) - 1) \cdot (N+1) + 1] \\ &= \frac{1}{N} \cdot [(N+1) \cdot \log_2(N+1) - N] \\ &\approx \log_2(N+1) - 1 \text{ für große } N \end{aligned}$$

Im Mittel verursacht binäres Suchen also nur etwa eine Kosteneinheit weniger als im schlechtesten Fall.

Divide & Conquer

Entwurfsprinzip:

- **Divide** the problem into subproblems.
- **Conquer** the subproblems by solving them recursively.
- **Combine** subproblem solutions.

Beispiele:

- Binäre Suche
- Potenzieren einer Zahl
- Matrix-Multiplikation
- Quicksort

Divide & Conquer: Potenzieren einer Zahl

Problem: Berechne x^n für ein $n \in \mathbb{N}$.

- Einfacher Algorithmus:

```
erg := 1
for i := 1 to n do
    erg := erg * x
```

  → Laufzeit: $\Theta(n)$ Multiplikationen

- Divide & Conquer: **???**

Divide & Conquer: Matrix-Multiplikation

Eingabe: zwei $n \times n$-Matrizen A und B
Ausgabe: $C = A \cdot B$

Es gilt:

$$\begin{pmatrix} c_{11} & c_{12} & \cdots & c_{1n} \\ c_{21} & c_{22} & \cdots & c_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ c_{n1} & c_{n2} & \cdots & c_{nn} \end{pmatrix} = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{pmatrix} \cdot \begin{pmatrix} b_{11} & b_{12} & \cdots & b_{1n} \\ b_{21} & b_{22} & \cdots & b_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n1} & b_{n2} & \cdots & b_{nn} \end{pmatrix}$$

mit

$$c_{ij} = \sum_{k=1}^{n} a_{ik} \cdot b_{kj}$$

Divide & Conquer: Matrix-Multiplikation (2)

Einfacher Algorithmus:

```
for i := 1 to n do
    for j := 1 to n do
        c[i][j] := 0
        for k := 1 to n do
            c[i][j] := c[i][j] + a[i][k] * b[k][j]
```

→ Laufzeit: $\Theta(n^3)$ Additionen/Multiplikationen

Divide & Conquer: Matrix-Multiplikation (3)

Aufteilen der $n \times n$-Matrizen in jeweils vier $\frac{n}{2} \times \frac{n}{2}$-Matrizen:

$$\left(\begin{array}{c|c} r & s \\ \hline t & u \end{array}\right) = \left(\begin{array}{c|c} a & b \\ \hline c & d \end{array}\right) \cdot \left(\begin{array}{c|c} e & f \\ \hline g & h \end{array}\right)$$

$$C = A \cdot B$$

mit

$$\begin{aligned} r &= ae + bg \\ s &= af + bh \\ t &= ce + dg \\ u &= cf + dh \end{aligned}$$

$\Rightarrow$ 8 Multiplikationen von $\frac{n}{2} \times \frac{n}{2}$-Matrizen
4 Additionen von $\frac{n}{2} \times \frac{n}{2}$-Matrizen

→ Laufzeit: $T(n) = 8 \cdot T(n/2) + 4 \cdot (n/2)^2 = \ldots = \Theta(n^3)$

Matrix-Multiplikation: Strassens Idee

$$\left(\begin{array}{c|c} r & s \\ \hline t & u \end{array}\right) = \left(\begin{array}{c|c} a & b \\ \hline c & d \end{array}\right) \cdot \left(\begin{array}{c|c} e & f \\ \hline g & h \end{array}\right)$$

mit

$$\begin{aligned} P_1 &= a \cdot (f - h) \\ P_2 &= (a + b) \cdot h \\ P_3 &= (c + d) \cdot e \\ P_4 &= d \cdot (g - e) \\ P_5 &= (a + d) \cdot (e + h) \\ P_6 &= (b - d) \cdot (g + h) \\ P_7 &= (a - c) \cdot (e + f) \end{aligned}$$

und

$$\begin{aligned} r &= P_5 + P_4 - P_2 + P_6 \\ s &= P_1 + P_2 \\ t &= P_3 + P_4 \\ u &= P_5 + P_1 - P_3 - P_7 \end{aligned}$$

$\rightarrow$ Laufzeit: $T(n) = 7T\left(\frac{n}{2}\right) + 18\left(\frac{n}{2}\right)^2 = \Theta(n^{\log_2 7}) \approx \Theta(n^{2.807})$

Matrix-Multiplikation: Strassens Idee (2)

Vergleich der Laufzeiten: (Annahme 1 GOp/s)

n	n^3	time	$n^{2.807}$	time	$n^{2.397}$	time
1.000	1e9	1 s	2.64e8	1 s	1.55e7	1 s
10.000	1e12	16 m	1.70e11	3 m	3.87e9	4 s
100.000	1e15	11 t	1.09e14	2 t	9.66e11	16 m
1.000.000	1e18	31 j	9.98e16	3 j	2.41e14	3 t

s: Sekunden
m: Minuten
t: Tage
j: Jahre

Rekursionsgleichungen

Master-Theorem:

$$T(n) = a \cdot T(n/b) + \Theta(n^k)$$

Unterscheide drei Fälle:

- für $a < b^k$ gilt: $T(n) = \Theta(n^k)$
- für $a = b^k$ gilt: $T(n) = \Theta(n^k \cdot \log(n))$
- für $a > b^k$ gilt: $T(n) = \Theta(n^{\log_b a})$

Rekursionsgleichungen (2)

Beispiele:

- Binäre Suche: $a = 1$, $b = 2$, $k = 0$

  $$a = b^k \;\rightarrow\; T(n) = \Theta(n^k \cdot \log(n)) = \Theta(\log(n))$$

- Matrix-Multiplikation: $a = 8$, $b = 2$, $k = 2$

  $$a > b^k \;\rightarrow\; T(n) = \Theta(n^{\log_b a}) = \Theta(n^3)$$

Algorithmen & Datenstrukturen

Sortieren

Quicksort

Einige Fakten:

- 1962 von C.A.R. Hoare veröffentlicht.
- divide & conquer Algorithmus
- eines der schnellsten allgemeinen Sortierverfahren
- in-situ: kein zusätzlicher Speicherplatz zur Speicherung von Datensätzen erforderlich (ausser einer konstanten Anzahl von Hilfsspeicherplätzen für Tauschoperationen)
- praxistauglich

Quicksort (2)

1. **Divide:** Wähle aus allen Werten einen beliebigen Wert p aus und teile die Folge in zwei Teilfolgen K und G auf:
   - K enthält Werte die kleiner oder gleich p sind,
   - G enthält Werte die größer oder gleich p sind.

2. **Conquer:** Sortiere K und G rekursiv
3. **Combine:** trivial

Noch offene Punkte:

- aufteilen der Folge in zwei Teilfolgen
- wählen des Pivot-Elements

Quicksort (3)

Pivot-Element: erstes Element der Teilfolge

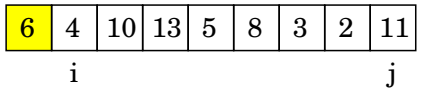
Aufteilen der Folge in zwei Teilfolgen:

```
partition(int l, int r)
    p := A[l],  i := l+1,  j := r
    repeat
        while (A[i] ≤ p) and (i < r) do i := i + 1
        while (A[j] ≥ p) and (j > l) do j := j - 1
        if i < j then swap(i, j)
    until j ≤ i
    swap(l, j)
    return j
```

→ Laufzeit: $\Theta(n)$ für n Elemente

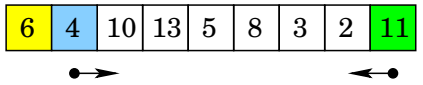
Quicksort (4)

Beispiel:



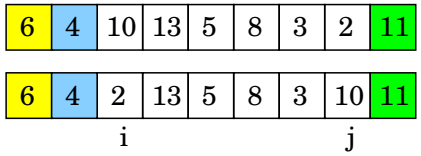
Quicksort (4)

Beispiel:



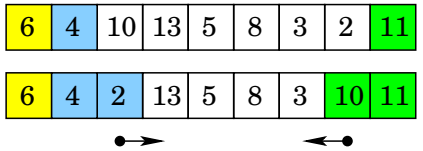
Quicksort (4)

Beispiel:



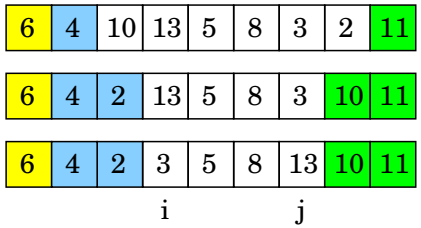
Quicksort (4)

Beispiel:



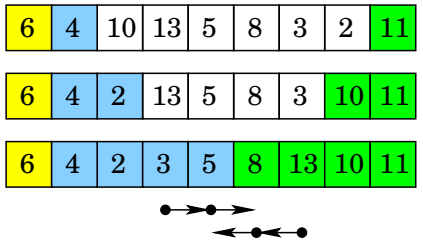
Quicksort (4)

Beispiel:



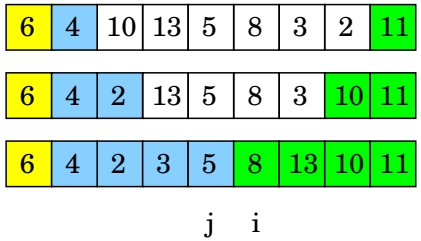
Quicksort (4)

Beispiel:



Quicksort (4)

Beispiel:



Quicksort (4)

Beispiel:

Quicksort (5)

Pseudo-Code:

```
quicksort(int l, int r)
    if l < r
        m := partition(l, r)
        quicksort(l, m-1)
        quicksort(m+1, r)
```

initial: quicksort(0, n-1)

Quicksort (6)

worst-case Analyse:

betrachte sortierte Folge: bei jedem rekursiven Aufruf ist die Teilfolge K leer und Teilfolge G ist um ein Element (dem Pivot-Element) kürzer geworden.

$$\begin{aligned} T(n) &= T(n-1) + \Theta(n) \\ &= T(n-2) + \Theta(n-1) + \Theta(n) \\ &= T(n-3) + \Theta(n-2) + \Theta(n-1) + \Theta(n) \\ &\vdots \\ &= T(1) + \Theta(2) + \ldots + \Theta(n-2) + \Theta(n-1) + \Theta(n) \\ &= \Theta\left(\sum_{k=1}^{n} k\right) = \Theta\left(\frac{n(n+1)}{2}\right) = \Theta(n^2) \end{aligned}$$

Quicksort (7)

best-case Analyse:

- die Folge wird bei jeder Aufteilung halbiert
- $T(n) = 2 \cdot T(n/2) + \Theta(n)$

Das Master-Theorem liefert für $a = 2$, $b = 2$ und $k = 1$:

$$T(n) = \Theta(n^k \cdot \log(n)) = \Theta(n \cdot \log(n))$$

Frage:

Welche Laufzeit hat Quicksort, wenn die Aufteilung immer in einem festen Verhältnis, z.B. $\frac{1}{10} : \frac{9}{10}$ erfolgt?

Quicksort (8)

Fazit:

- worst case: $T(n) \in \Theta(n^2)$
- average case: $T(n) \in \Theta(n \cdot \log(n))$
- best case: $T(n) \in \Theta(n \cdot \log(n))$

Problem: Laufzeit $O(n^2)$ bei stark vorsortierten Folgen

Lösung: Zufallsstrategie: wähle als Pivot-Element ein zufälliges Element aus $A[l...r]$ und vertausche es mit $A[l]$.

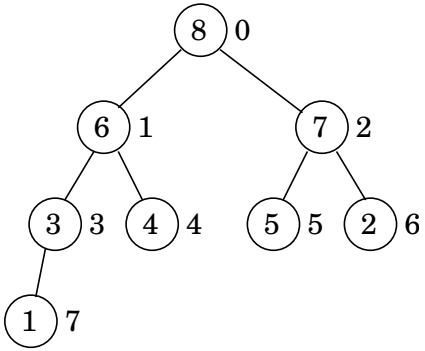
$\Rightarrow$ Laufzeit ist unabhängig von der zu sortierenden Folge

$\Rightarrow$ mittlere/erwartete Laufzeit: $\Theta(n \cdot \log(n))$

Heapsort

Heap: eine Folge $F = k_0, \ldots, k_n$ von Schlüsseln, so dass $k_i \geq k_{2i+1}$ und $k_i \geq k_{2(i+1)}$ gilt, sofern $2i+1 \leq n$ bzw. $2(i+1) \leq n$.

Beispiel: $F = 8, 6, 7, 3, 4, 5, 2, 1$



Heap-Eigenschaft ist erfüllt:

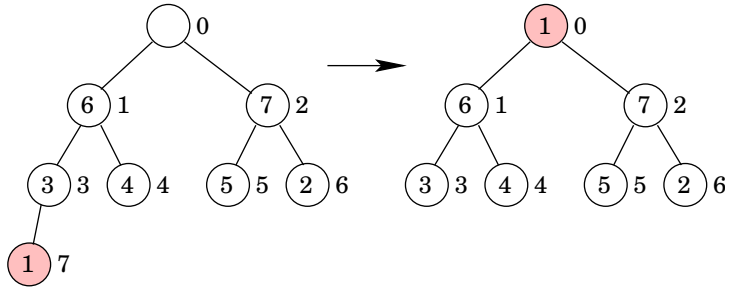
- $k_0 = 8 \geq k_1 = 6$ und $k_0 \geq k_2 = 7$
- $k_1 = 6 \geq k_3 = 3$ und $k_1 \geq k_4 = 4$
- $k_2 = 7 \geq k_5 = 5$ und $k_2 \geq k_6 = 2$
- $k_3 = 3 \geq k_7 = 1$

Heapsort (2)

Bestimmung des Maximums ist leicht: k_0 ist das Maximum.

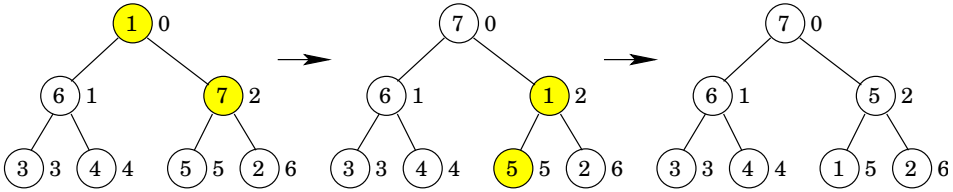
Das nächst kleinere Element wird bestimmt, indem das Maximum aus F entfernt wird und die Restfolge wieder zu einem Heap transformiert wird:

1. Setze den Schlüssel mit dem größten Index an die erste Position. ⇒ Heap-Eigenschaft verletzt!



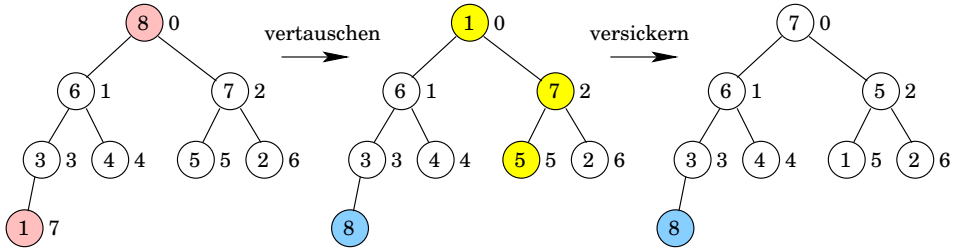
Heapsort (3)

2. Schlüssel versickern lassen, indem er immer mit dem größten seiner Nachfolger getauscht wird, bis entweder beide Nachfolger kleiner sind oder der Schlüssel unten angekommen ist.

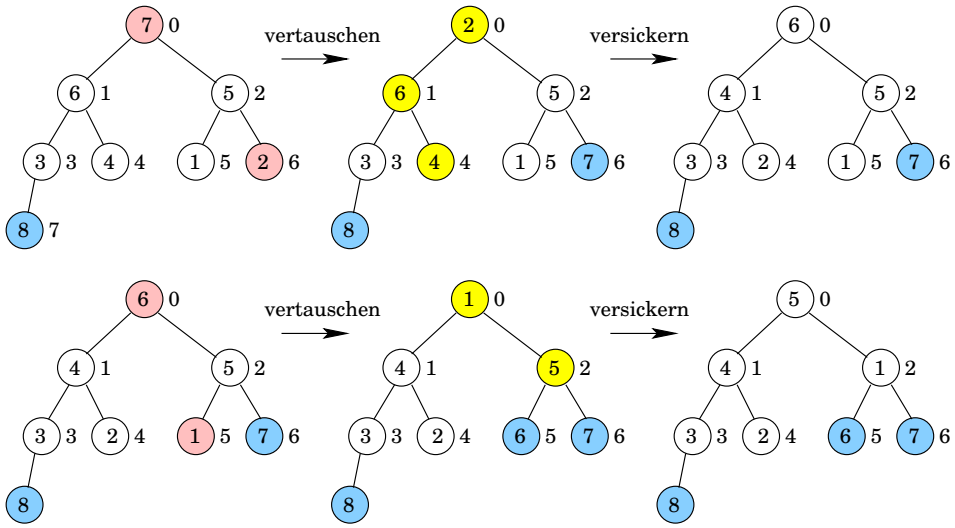


Heapsort (4)

Die Datensätze können sortiert werden, indem das jeweils aus dem Heap entfernte Maximum an die Stelle desjenigen Schlüssels geschrieben wird, der nach dem Entfernen des Maximums nach k_0 übertragen wird.



Heapsort (5)



usw.

Heapsort (6)

Analyse:

- Es erfolgen $n-1$ Vertauschungen außerhalb der Funktion `versickern`.
- Innerhalb von `versickern` wird ein Schlüssel wiederholt mit einem seiner Nachfolger vertauscht, wobei der Datensatz jeweils eine Stufe tiefer wandert.
- Heap mit n Datensätzen hat $\lceil \log_2(n+1) \rceil$ viele Ebenen.

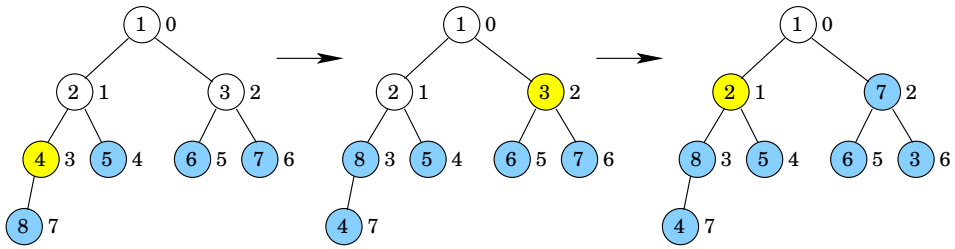
$\Rightarrow$ obere Schranke von $O(n \cdot \log(n))$ Vertauschungen

Heapsort (7)

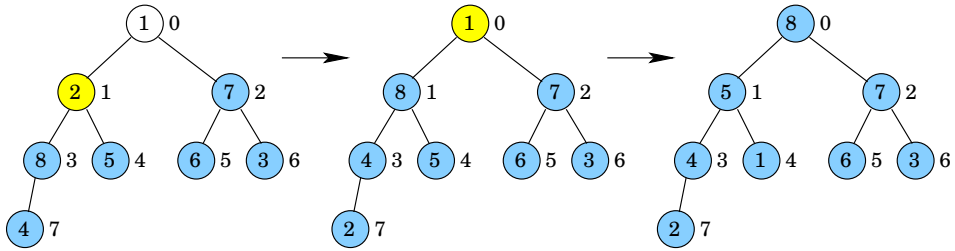
Wie wird die Anfangsfolge in einen Heap transformiert?

Idee: Lasse die Schlüssel $k_{n/2-1}, \ldots, k_0$ versickern. Dadurch werden schrittweise immer größere Heaps aufgebaut, bis letztlich ein Heap übrig bleibt.

Beispiel: $F = 1, 2, 3, 4, 5, 6, 7, 8$



Heapsort (8)



Analyse: aufbauen des initialen Heaps ist in linearer Zeit möglich, weil ...

Heapsort (9)

Damit ergibt sich insgesamt: Laufzeit $O(n \cdot \log(n))$

Anmerkungen:

- eine Vorsortierung der Eingabefolge schadet und nützt der Sortierung nichts
- Heapsort benötigt nur konstant viel zusätzlichen Speicherplatz (in-situ Sortierverfahren)

Untere Schranke

allgemeine Sortierverfahren: Algorithmen, die nur Vergleichsoperationen zwischen Schlüsseln verwenden.

Satz: Jedes allgemeine Sortierverfahren benötigt zum Sortieren von n verschiedenen Schlüsseln im schlechtesten Fall und im Mittel wenigstens $\Omega(n \cdot \log(n))$ Schlüsselvergleiche.

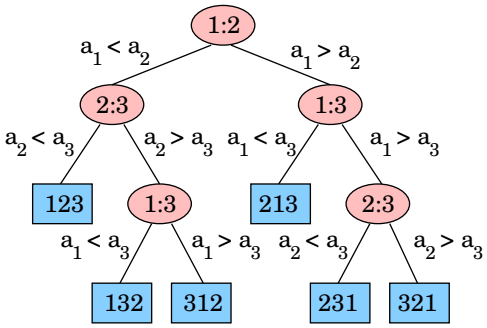
In [1] wird Sortierverfahren vorgestellt, das arithmetische Operationen und die Floor-Funktion benutzt, und das im Mittel eine Laufzeit von $O(n)$ hat.

---

[1] W. Dobosiewicz: Sorting by distributive partitioning. Information Processing Letters, 7(1), 1978

Untere Schranke (2)

Entscheidungsbaum:



- innere Knoten sind mit $i : j$ beschriftet für $i, j \in \{1, 2, \ldots, n\}$
- linker Teilbaum enthält nachfolgende Vergleiche, falls $a_i < a_j$
- rechter Teilbaum: nachfolgende Vergleiche, falls $a_i > a_j$
- jedes Blatt: Permutation $(\pi(1), \pi(2), \ldots, \pi(n))$ bezeichnet die Sortierung $a_{\pi(1)} \leq a_{\pi(2)} \leq \cdots \leq a_{\pi(n)}$

Untere Schranke (3)

Ein Entscheidungsbaum kann jedes allgemeine Sortierverfahren modellieren:

- jeweils ein Baum für jede Eingabegröße n
- Laufzeit des Algorithmus = Länge des gewählten Pfades
- worst-case Laufzeit = Höhe des Baums

Untere Schranke (4)

worst-case Analyse:

- es gibt $n!$ verschiedene Permutationen über n Zahlen
- ein Entscheidungsbaum hat mindestens $n!$ Blätter
- ein Binärbaum der Höhe h hat maximal $2^h - 1$ Blätter

$\Rightarrow 2^h \geq n!$

$$\begin{aligned} h &\geq \log(n!) && \text{log ist monoton steigend} \\ &\geq \log((n/e)^n) && \text{Stirling-Formel} \\ &= \log(n^n) - \log(e^n) \\ &= n \cdot \log(n) - n \cdot \log(e) \\ &\in \Omega(n \cdot \log(n)) \end{aligned}$$

Untere Schranke (5)

average-case Analyse: Beweis durch Widerspruch

- **Behauptung:** Die mittlere Tiefe eines Entscheidungsbaums mit k Blättern ist wenigstens $\log_2(k)$.
- **Annahme:** Es existiert ein Entscheidungsbaum mit k Blättern dessen mittlere Tiefe kleiner als $\log_2(k)$ ist.

Sei T ein solcher Baum, der unter all diesen Bäumen der kleinste ist. T habe k Blätter. Dann gilt:

- T hat einen linken Teilbaum T_1 mit k_1 Blättern
- T hat einen rechten Teilbaum T_2 mit k_2 Blättern
- es gilt $k_1 < k$, $k_2 < k$ und $k_1 + k_2 = k$

Untere Schranke (6)

Da T der kleinste Baum ist, für den die Annahme gilt, erhalten wir für T_1 und T_2:

$$\text{mittlere Tiefe}(T_1) \geq \log_2(k_1)$$

$$\text{mittlere Tiefe}(T_2) \geq \log_2(k_2)$$

Jedes Blatt in T_1 bzw. T_2 auf Tiefe t hat in T die Tiefe $t+1$. Also gilt insgesamt:

$$\begin{aligned} \mathrm{mT}(T) &= \frac{k_1}{k}(\mathrm{mT}(T_1)+1) + \frac{k_2}{k}(\mathrm{mT}(T_2)+1) \\ &\geq \frac{k_1}{k}(\log_2(k_1)+1) + \frac{k_2}{k}(\log_2(k_2)+1) \\ &= \frac{1}{k}(k_1 \cdot \log_2(2k_1) + k_2 \cdot \log_2(2k_2)) \end{aligned}$$

Untere Schranke (7)

Unter der Nebenbedingung $k_1 + k_2 = k$ hat die Funktion

$$\mathrm{mT}(T) = \frac{1}{k}(k_1 \cdot \log_2(2k_1) + k_2 \cdot \log_2(2k_2))$$

ein Minimum bei $k_1 = k_2 = k/2$. Damit gilt

$$\mathrm{mT}(T) \geq \frac{1}{k}\left(\frac{k}{2}\log_2(k) + \frac{k}{2}\log_2(k)\right) = \log_2(k)$$

im Widerspruch zur Annahme.

Jeder Entscheidungsbaum zur Sortierung von n Zahlen hat mindestens $n!$ Blätter, daher gilt:

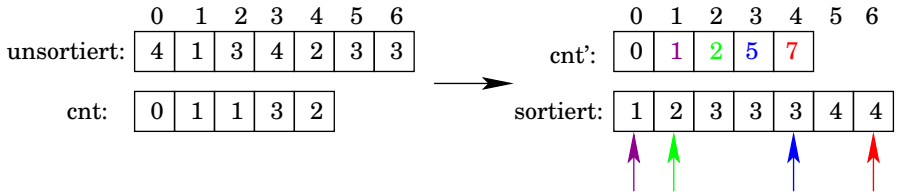
$$\mathrm{mT}(T) \geq \log_2(n!) \geq \log_2((n/e)^n) \in \Omega(n \log(n))$$

Counting Sort

Sortierverfahren ohne Schlüsselvergleiche!

Eingabe: $a[0...N]$ mit $a[i] \in \{0,\ldots,k\}$
Ausgabe: $b[0...N]$ sortiert
Hilfsspeicher: $cnt[0...k]$ zum Zählen

Idee: Zähle die Häufigkeiten der Zahlen $0,\ldots,k$ in der zu sortierenden Liste und berechne daraus die Position der jeweiligen Zahl in der sortierten Liste.



Counting Sort (2)

Pseudo-Code:

```
for i := 0 to k-1 do                    /* init */
    cnt[i] := 0

for j := 0 to n-1 do                    /* count */
    cnt[a[j]] := cnt[a[j]] + 1

for i := 1 to k-1 do                    /* collect */
    cnt[i] := cnt[i] + cnt[i-1]

for j := n-1 downto 0 do                /* rearrange */
    b[cnt[a[j]] - 1] := a[j]
    cnt[a[j]] := cnt[a[j]] - 1
```


Counting Sort (3a)

	0	1	2	3	4	5	6	7	8	9	10	11	12
a:	3	6	7	5	3	5	6	2	9	1	2	7	3

	0	1	2	3	4	5	6	7	8	9
cnt:	0	1	2	3	0	2	2	2	0	1

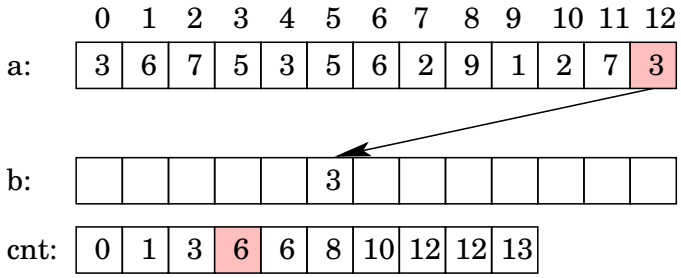
Counting Sort (3b)

	0	1	2	3	4	5	6	7	8	9	10	11	12
a:	3	6	7	5	3	5	6	2	9	1	2	7	3

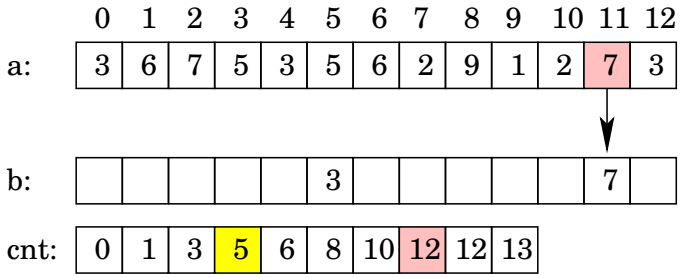
	0	1	2	3	4	5	6	7	8	9
cnt:	0	1	2	3	0	2	2	2	0	1

	0	1	2	3	4	5	6	7	8	9
cnt:	0	1	3	6	6	8	10	12	12	13

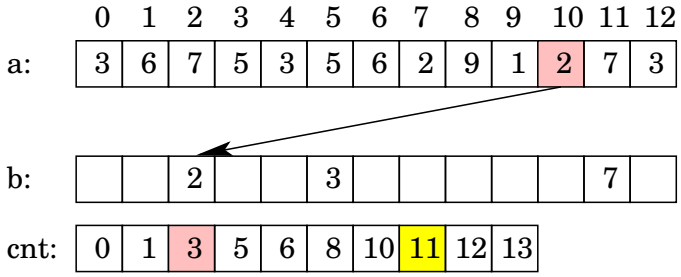
Counting Sort (3c)



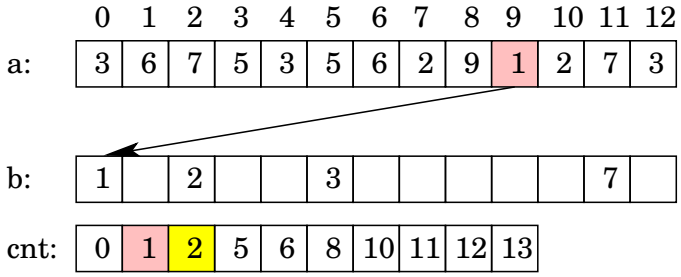
Counting Sort (3d)



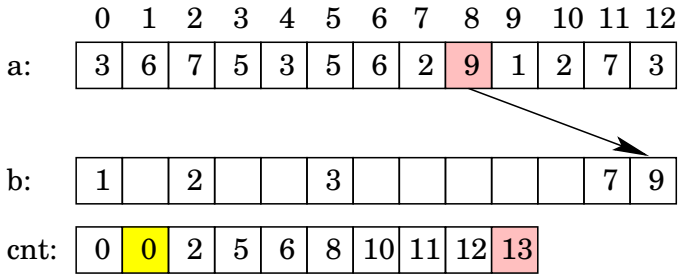
Counting Sort (3e)



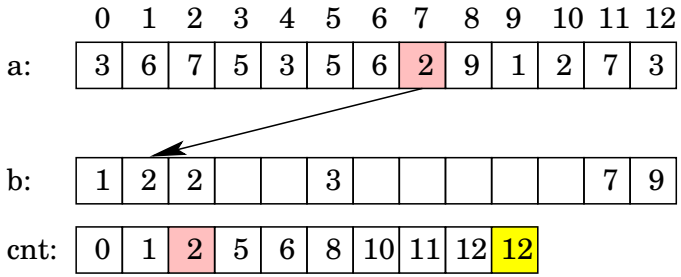
Counting Sort (3f)



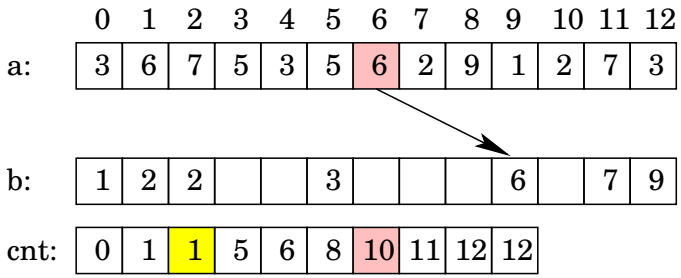
Counting Sort (3g)



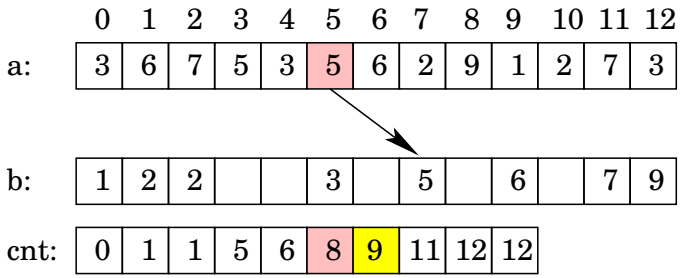
Counting Sort (3h)



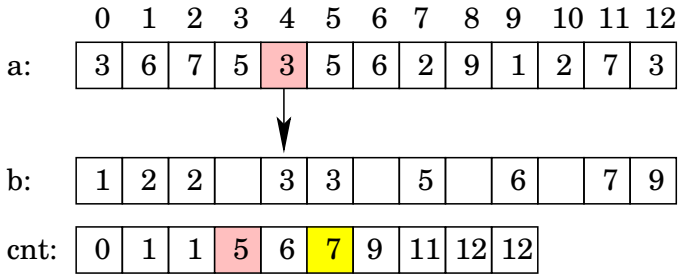
Counting Sort (3i)



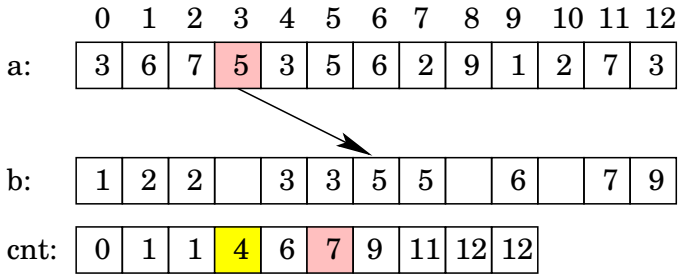
Counting Sort (3j)



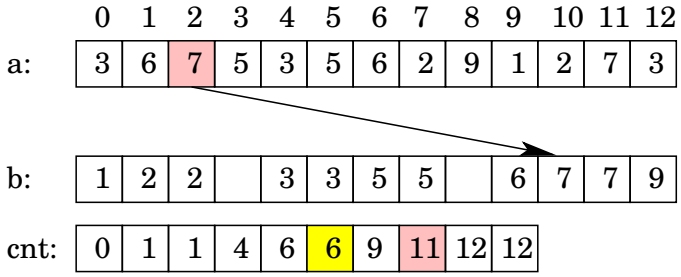
Counting Sort (3k)



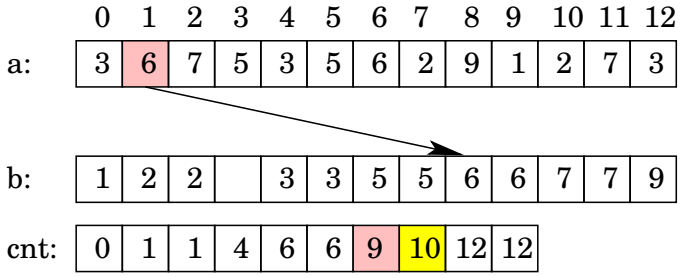
Counting Sort (3l)



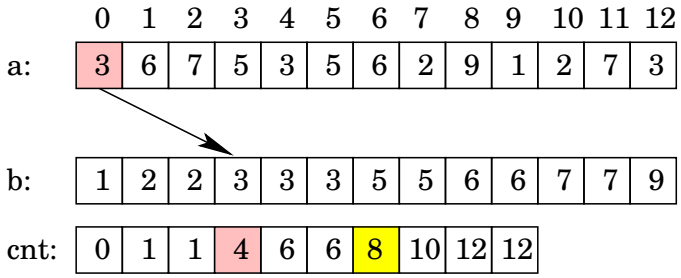
Counting Sort (3m)



Counting Sort (3n)



Counting Sort (3o)

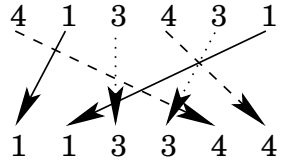


Counting Sort (4)

Laufzeit:

init	$\Theta(k)$
count	$\Theta(n)$
collect	$\Theta(k)$
rearrange	$\Theta(n)$
	$\Theta(n+k)$

Counting Sort ist ein **stabiles Sortierverfahren**: gleiche Elemente stehen nach dem Sortieren in der gleichen Reihenfolge wie vor dem Sortieren.



Radix Sort

Problem: Counting Sort ist ineffizient bzgl. Laufzeit und Speicherplatz, wenn der Wertebereich groß ist im Vergleich zur Anzahl der Zahlen: $\Theta(n+k) = \Theta(n^2)$ für $k = n^2$

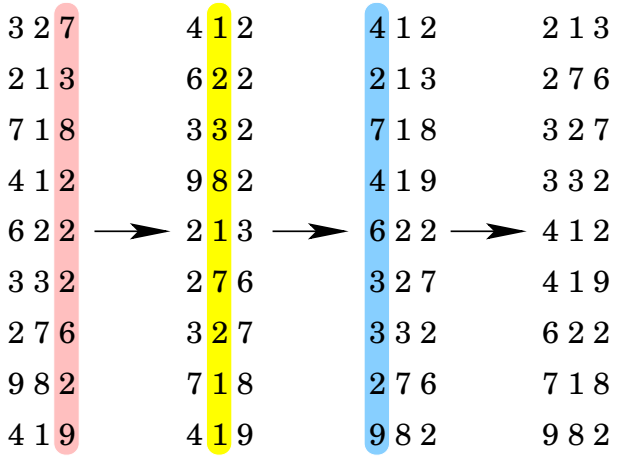
Beispiel: es sollen 100.000 Datensätze mit einem 32Bit-Schlüssel sortiert werden, also $k = 2^{32} \approx 4.300.000.000$

- $n + k \approx 4.300.000.000$
- $n \cdot \log(n) \approx 1.600.000$

Lösung: Sortiere die Zahlen anhand der einzelnen Ziffern, beginnend mit der niederwertigsten Stelle. Verwende ein stabiles Sortierverfahren wie Counting Sort. Falls nötig, müssen führende Nullen eingefügt werden.

Radix Sort (2)

Beispiel:



Radix Sort (3)

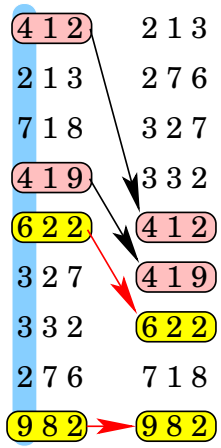
Korrektheit: Induktion über die betrachtete Position

I.A. nach der ersten Sortierphase sind die Zahlen bezüglich der Position 0 sortiert

I.V. die Zahlen sind bezüglich der $t-1$ niederwertigsten Ziffern sortiert

I.S. Sortiere nach Ziffer t:

- zwei Zahlen, die sich an Position t unterscheiden, sind richtig sortiert
- zwei Zahlen, die an Position t dieselbe Ziffer haben, sind nach I.V. richtig sortiert und bleiben aufgrund des stabilen Sortierverfahrens sortiert



Radix Sort (4)

Laufzeit:

- fasse jeweils r bits zu einer Ziffer zusammen
- bei einer Wortlänge von b bit müssen b/r Phasen durchlaufen werden

Frage: Wie groß muss r im Verhältnis zu b sein, um eine gute Laufzeit zu erzielen?

- Counting Sort hat Laufzeit $\Theta(n+k)$. Hier: $\Theta(n+2^r)$
- bei b/r Phasen erhalten wir: $\Theta(b/r \cdot (n+2^r))$
- Extremwert bestimmen: bei welchem Wert von r wird die Laufzeit minimal?

Praxis: 32bit-Wörter in Gruppen von 8bit $\Rightarrow$ 4 Phasen

Algorithmen & Datenstrukturen

Suchen

Exponentielle Suche

Gesucht wird ein Element mit Schlüssel k. Die exponentielle Suche eignet sich zum Suchen in sortierten, linearen Listen, deren Länge nicht bekannt ist.

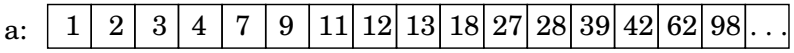
Bestimme in exponentiell wachsenden Schritten einen Bereich, in dem der Schlüssel liegen muss:

```
i = 1;
while (k > a[i].key)
    i *= 2;
```

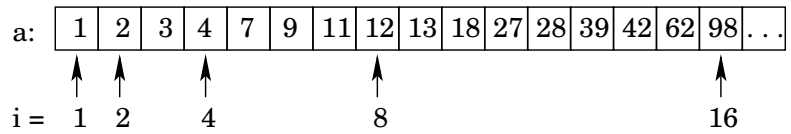
Danach gilt: `a[i/2].key` $<$ `k` $\leq$ `a[i].key`
Dieser Bereich wird mittels binärer Suche durchsucht.

Exponentielle Suche: Beispiel

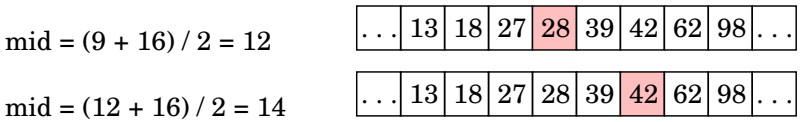
Wir suchen den Schlüssel $k = 42$ in folgender Liste:



(1) bestimme den Bereich, in dem k liegen muss:



(2) binäre Suche im Bereich `a[i/2+1] ... a[i]`:



Exponentielle Suche: Laufzeit

Voraussetzung: Werte sind paarweise verschieden (dies ist z.B. dann gegeben, wenn Schlüsselwerte gesucht werden, Telefonnummer, Matrikelnummer, Personalnummer, usw.)

$\Rightarrow$ Schlüssel wachsen mindestens so stark wie die Indices

$\Rightarrow$ die Bereichsbestimmung erfolgt in $\log_2(k)$ Schritten

$\Rightarrow$ der zu durchsuchende Bereich hat höchstens k Zahlen und kann in Zeit $\log_2(k)$ durchsucht werden

insgesamt ergibt sich also eine Laufzeit von $\Theta(\log(k))$

Falls die Folge auch gleiche Werte enthalten darf, ist keine Laufzeitabschätzung möglich.

Interpolations-Suche

Idee: Schätze die Position des Elements mit Schlüssel k.

Beispiel: Im Telefonbuch steht der Name `Zimmermann` weit hinten, wohingegen `Brockmann` eher am Anfang steht.

Sei l linke Grenze, r rechte Grenze des Suchbereichs:

- bei der binären Suche wurde der Index des nächsten zu inspizierenden Elements bestimmt als

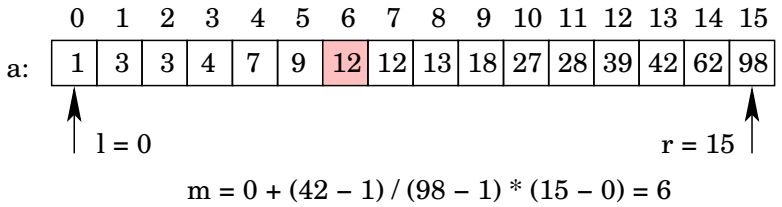
$$m \;=\; l + \frac{1}{2} \cdot (r - l).$$

- bei der Interpolationssuche inspiziere als nächstes das Element auf Position

$$m \;=\; l + \frac{k - a[l].key}{a[r].key - a[l].key} \cdot (r - l).$$

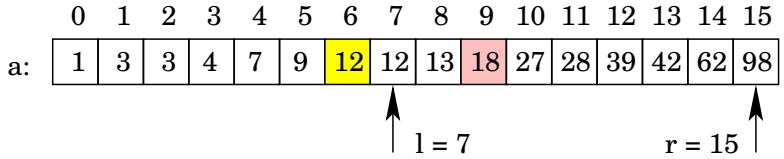
Interpolations-Suche: Beispiel (a)

gesucht: $k = 42$



Interpolations-Suche: Beispiel (b)

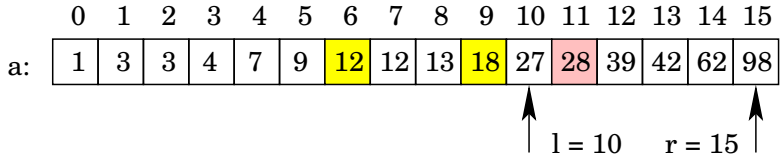
gesucht: $k = 42$



m = 7 + (42 – 12) / (98 – 12) * (15 – 7) = 9

Interpolations-Suche: Beispiel (c)

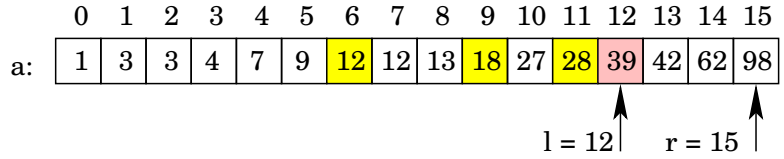
gesucht: $k = 42$



m = 10 + (42 – 27) / (98 – 27) * (15 – 10) = 11

Interpolations-Suche: Beispiel (d)

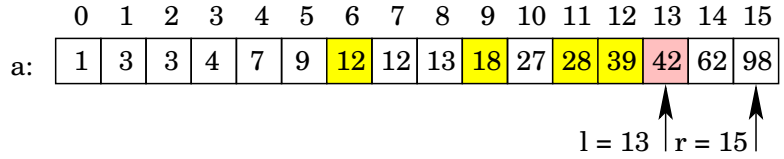
gesucht: $k = 42$



m = 12 + (42 – 39) / (98 – 39) * (15 – 12) = 12

Interpolations-Suche: Beispiel (e)

gesucht: $k = 42$



m = 13 + (42 – 42) / (98 – 42) * (15 – 13) = 13

Interpolations-Suche: Laufzeit

Im Mittel werden $\log(\log(N)) + 1$ Schlüsselvergleiche ausgeführt [1], aber dieser Vorteil geht durch die auszuführenden arithmetischen Operationen oft verloren.

Im worst-case hat das Verfahren eine lineare Laufzeit, also $\Theta(n)$ bei n Schlüsselwerten.

Beispiel: $k = 10$ und $F = 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 1000$

---

[1] A.C. Yao and F.F. Yao: The complexity of searching an ordered random table. Proceedings of the Symposium on Foundations of Computer Science, 1976

Hash-Verfahren

Schlüsselsuche durch Berechnung der Array-Indices!

Idee:

- bei einer Menge von Werten $K \subseteq \{0, 1, \ldots, m-1\}$
- verwende ein Array $A[0 \ldots m-1]$
- und speichere die Schlüssel wie folgt:

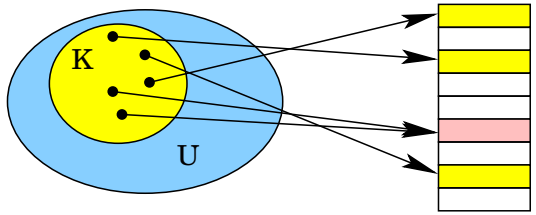
$$A[k] = \begin{cases} x & \text{falls } x.key \in K \text{ und } x.key = k \\ nil & \text{sonst} \end{cases}$$

$\Rightarrow$ suchen, einfügen und löschen in $\Theta(1)$ Schritten

Problem: Wertebereich kann sehr groß sein
8-stellige Namen $\rightarrow$ mehr als 208.000.000.000 Werte

Hash-Verfahren (2)

Lösung: verwende eine Hash-Funktion h, um den Wertebereich U, speziell die Menge der Schlüsselwerte K, auf die Zahlen $0, \ldots, m-1$ abzubilden, also $h : U \rightarrow \{0, \ldots, m-1\}$.



Hash-Funktion i.Allg. nicht injektiv: verschiedene Schlüssel werden auf dieselbe Hash-Adresse abgebildet $\rightarrow$ Kollision

Hash-Funktion wird zum Plazieren und Suchen verwendet, muss einfach/effizient zu berechnen sein

Hash-Verfahren (3)

Schlüssel, die nicht als Zahlen interpretiert werden können, müssen vorher geeignet umgerechnet werden.

Beispiel: Zeichenketten der Länge 3

- Die ASCII-Darstellung wird als Binärzahl interpretiert.
- Interpretiere die Binärzahlen als Ziffern einer Zahl zur Basis 256.

Input	k_1	k_2	k_3	k	$h(k) = k \bmod 11$
`i`	0	0	105	105	6
`i2`	0	105	50	26930	2
`ii`	0	105	105	26985	2
`iii`	105	105	105	6908265	1
`i_2`	105	95	50	6905650	4

Hash-Verfahren (4)

Hash-Verfahren muss zwei Forderungen genügen:

1. es sollen möglichst wenige Kollisionen auftreten
2. Adress-Kollisionen müssen effizient aufgelöst werden

Wahl der Hash-Funktion:

- soll die zu speichernden Datensätze möglichst gleichmäßig auf den Speicherbereich verteilen, um Adress-Kollisionen zu vermeiden
- Häufungen in der Schlüsselverteilung sollen sich nicht auf die Verteilung der Adressen auswirken
- es gilt: wenn eine Hash-Funktion $\sqrt{\pi n/2}$ Schlüssel auf eine Tabelle der Größe n abbildet, dann gibt es fast sicher eine Kollision (für $n = 365$ ist $\sqrt{\pi n/2} \approx 23$)

Hash-Funktion: Division-Rest-Methode

Der Schlüssel wird ganzzahlig durch die Länge der Hash-Tabelle dividiert. Der Rest wird als Index verwendet:

$$h(k) = k \bmod m$$

zu beachten:

- m soll keinen kleinen Teiler haben!
- m soll keine Potenz der Basis des Zahlensystems sein! Beispiel: für $m = 2^r$ hängt der Hash-Wert nur von den letzen r Bit ab

$\Rightarrow$ wähle m als Primzahl, die nicht nah an einer Potenz der Basis des Zahlensystems liegt. Beispiel: $m = 761$, **aber nicht:** $m = 509$ (nah an 2^9) oder $m = 997$ (nah an 10^3)

Hash-Funktion: Multiplikative Methode

Sei $m = 2^r$ eine Zweierpotenz. Bei einer Wortgröße w wähle eine Zahl a so, dass $2^{w-1} < a < 2^w$ ist.

$$h(k) = (a \cdot k \bmod 2^w) >> (w - r)$$

Anmerkungen:

- Wähle a nicht zu nah an 2^w
- Modulo-Operation und Rechts-Shift ist schnell
- gute Ergebnisse für $a \approx \frac{\sqrt{5}-1}{2} \cdot 2^w$ (goldener Schnitt)

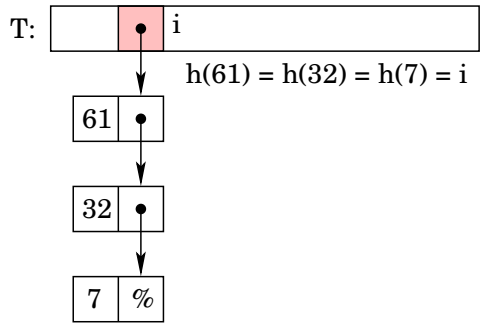
Beispiel: Für $w = 8$, $r = 3$, $a = 191$ und $k = 23$ erhalten wir

	10111111	191
*	00010111	23
10001001	01001	4393

Hash-Verfahren: Verkettung der Überläufer

Probleme treten auf

- beim Einfügen, wenn die berechnete Hash-Adresse nicht leer ist
- bei der Suche, wenn der berechnete Platz ein anderes Element enthält



Kollisionsauflösung: Die Überläufer können in einer linearen Liste verkettet werden, die an den Hash-Tabelleneintrag angehängt wird.

Verkettung der Überläufer: Bewertung

Die durchschnittliche Anzahl der Einträge in $h(k)$ ist N/M, wenn N Einträge durch die Hash-Funktion gleichmäßig auf M Listen verteilt werden. **Belegungsfaktor:** $\alpha = \frac{N}{M}$

Mittlere Laufzeit bei erfolgreicher Suche:

- beim Einfügen des j-ten Schlüssels ist die durchschnittliche Listenlänge $\frac{j-1}{M}$
- bei einer späteren Suche nach dem j-ten Schlüssel betrachten wir also im Durchschnitt $1+\frac{j-1}{M}$ Einträge, wenn stets am Listenende eingefügt wird und keine Elemente gelöscht wurden

$$\overline{C} = \frac{1}{N}\sum_{j=1}^{N}\left(1+\frac{j-1}{M}\right) = 1+\frac{N-1}{2M} \approx 1+\frac{N}{2M}$$

Verkettung der Überläufer: Bewertung (2)

in der Regel gilt $N \in O(M)$ (die Größe der Hash-Tabelle ist ungefähr so groß wie die Anzahl der Datensätze)

$$\Rightarrow \quad \alpha = \frac{N}{M} = \frac{O(M)}{M} \in O(1)$$

zum Vergleich: binäre Suche hat Laufzeit $\Theta(\log(N))$ plus Aufwand $\Theta(N\log(N))$ zum Sortieren der Datensätze

Problem: verkettete Listen sind umständlich zu Programmieren und dynamische Speicheranforderung ist teuer.

Lösung: speichere alle Datensätze innerhalb des Arrays

offene Hash-Verfahren

Idee: Speichere die Überläufer in der Hash-Tabelle, nicht in zusätzlichen Listen

- ist die Hash-Adresse $h(k)$ belegt, so wird systematisch eine Ausweichposition gesucht
- die Folge der zu betrachtenden Speicherplätze für einen Schlüssel nennt man **Sondierungsfolge**
- die Hash-Funktion hängt vom Schlüssel und von der Anzahl durchgeführter Plazierungsversuche ab:

$$h : U \times \{0,\ldots,m-1\} \to \{0,\ldots,m-1\}$$

- die Sondierungsfolge muss eine Permutation der Zahlen $0,\ldots,m-1$ sein, damit alle Einträge der Hash-Tabelle genutzt werden können

offene Hash-Verfahren (2)

Anmerkungen:

- beim Einfügen und Suchen wird dieselbe Sondierungsfolge durchlaufen
- beim Löschen wird der Datensatz nicht gelöscht, sondern nur als gelöscht markiert (der Wert wird ggf. bei späterem Einfügen überschrieben)
- je voller die Tabelle wird, umso schwieriger wird das Einfügen neuer Schlüssel

lineares Sondieren

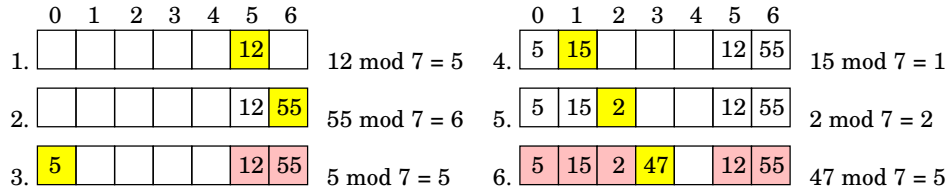
zu einer gegebenen Hash-Funktion $h'(k)$ sei

$$h(k,i) = (h'(k) + i) \bmod m.$$

Beispiel: betrachte das Einfügen der Schlüssel

$$12, 55, 5, 15, 2, 47$$

für $M = 7$ und $h'(k) = k \bmod 7$



quadratisches Sondieren

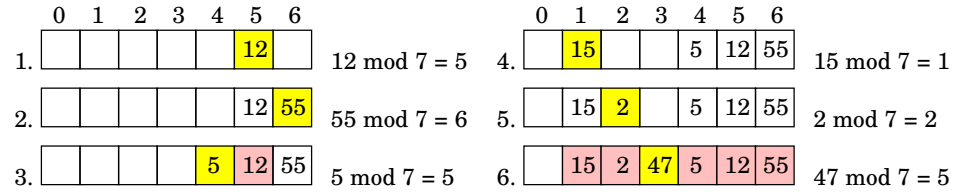
zu einer gegebenen Hash-Funktion $h'(k)$ sei

$$h(k,i) = (h'(k) + (-1)^i \cdot \lceil i/2 \rceil^2) \bmod m.$$

Beispiel: betrachte das Einfügen der Schlüssel

$$12, 55, 5, 15, 2, 47$$

für $M = 7$ und $h'(k) = k \bmod 7$



double Hashing

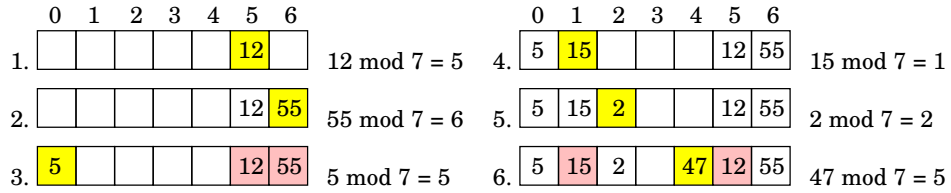
zu zwei gegebenen Hash-Funktionen $h_1(k)$ und $h_2(k)$ sei

$$h(k,i) = (h_1(k) + i \cdot h_2(k)) \bmod m.$$

Beispiel: betrachte das Einfügen der Schlüssel

$$12, 55, 5, 15, 2, 47$$

für $M = 7$, $h_1(k) = k \bmod 7$ und $h_2 = 1 + k \bmod 5$



Algorithmen & Datenstrukturen

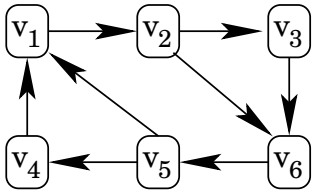
Graphalgorithmen

Grundlagen

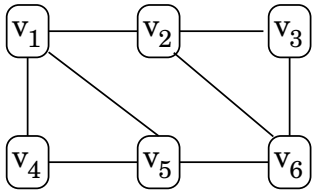
Ein **gerichteter Graph** $G = (V, E)$ besteht aus

- einer endlichen Menge von **Knoten** $V = \{v_1, \ldots, v_n\}$ und
- einer Menge von gerichteten **Kanten** $E \subseteq V \times V$

Bei einem **ungerichteten Graphen** $G = (V, E)$ sind die Kanten ungeordnete Paare: $E \subseteq \{\{u, v\} \mid u, v \in V, u \neq v\}$



gerichteter Graph



ungerichteter Graph

Motivation

Graphen verwenden, wo Sachverhalt darstellbar ist durch

- eine Menge von Objekten (Entitäten)
- Beziehungen zwischen den Objekten

Beispiele:

- **Routenplanung:** Städte sind durch Straßen verbunden
- **Kursplanung:** Kurse setzen andere Kurse voraus
- **Produktionsplanung:** Produkte werden aus Teilen zusammengesetzt
- **Schaltkreisanalyse:** Bauteile sind durch elektrische Leitungen verbunden
- **Spiele:** Objekte: Status, Beziehungen: Spielzüge

Begriffe: gerichtete Graphen

Sei $G = (V, E)$ ein gerichteter Graph und $u, v \in V$.

- Sei $e = (u, v)$ eine Kante. Knoten u ist der **Startknoten**, v der **Endknoten** von e. Die Knoten u und v sind **adjazent**, Knoten u bzw. v und Kante e sind **inzident**.
- $indeg(u)$: **Eingangsgrad** von u, Anzahl der in u einlaufenden Kanten. $outdeg(u)$: **Ausgangsgrad** von u, Anzahl der aus u auslaufenden Kanten.
- Ein Graph $G' = (V', E')$ ist ein **Teilgraph** von G, geschrieben $G' \subseteq G$, falls $V' \subseteq V$ und $E' \subseteq E$.
- $p = (v_0, v_1, \ldots, v_k)$ ist ein **gerichteter Weg** in G der Länge k von Knoten u nach Knoten w, falls gilt: $v_0 = u$, $v_k = w$ und $(v_{i-1}, v_i) \in E$ für $1 \leq i \leq k$.

  p ist **einfach**, wenn kein Knoten mehrfach vorkommt.

Begriffe: ungerichtete Graphen

Sei $G = (V, E)$ ein ungerichteter Graph und $u, v \in V$.

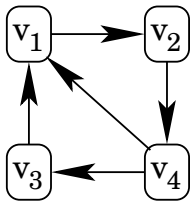
- Sei $e = \{u, v\}$ eine Kante. Die Knoten u und v sind **adjazent**, Knoten u bzw. v und Kante e sind **inzident**. Knoten u und v sind die **Endknoten** der Kante e.
- Der **Knotengrad** von u, geschrieben $deg(u)$, ist die Anzahl der zu u inzidenten Kanten.
- Ein Graph $G' = (V', E')$ ist ein **Teilgraph** von G, geschrieben $G' \subseteq G$, falls $V' \subseteq V$ und $E' \subseteq E$.
- $p = (v_0, v_1, \ldots, v_k)$ ist ein **ungerichteter Weg** in G der Länge k von Knoten u nach Knoten w, falls gilt: $v_0 = u$, $v_k = w$ und $\{v_{i-1}, v_i\} \in E$ für $1 \leq i \leq k$.

Speicherung von Graphen: Adjazenz-Matrix

Sei n die Anzahl der Knoten in Graph $G = (V, E)$. Die Adjazenz-Matrix für G ist eine $n \times n$-Matrix $A_G = (a_{ij})$ mit

$$a_{ij} = \begin{cases} 0, & \text{falls } (v_i, v_j) \notin E, \\ 1, & \text{falls } (v_i, v_j) \in E. \end{cases}$$

Beispiel:



gerichtet:

A	1	2	3	4
1	0	1	0	0
2	0	0	0	1
3	1	0	0	0
4	1	0	1	0

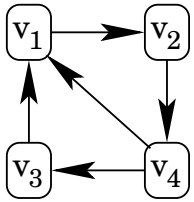
ungerichtet:

A	1	2	3	4
1	0	1	1	1
2	1	0	0	1
3	1	0	0	1
4	1	1	1	0

Speicherung von Graphen: Adjazenz-Liste

Für jeden Knoten v eines Graphen $G = (V, E)$ werden in einer doppelt verketteten Liste $Adj[v]$ alle von v ausgehenden Kanten gespeichert.

Beispiel:



gerichtet:
$Adj[v_1] = \{v_2\}$
$Adj[v_2] = \{v_4\}$
$Adj[v_3] = \{v_1\}$
$Adj[v_4] = \{v_1, v_3\}$

ungerichtet:
$Adj[v_1] = \{v_2, v_3, v_4\}$
$Adj[v_2] = \{v_1, v_4\}$
$Adj[v_3] = \{v_1, v_4\}$
$Adj[v_4] = \{v_1, v_2, v_3\}$

Speicherung von Graphen: Vergleich

Sei $G = (V, E)$ ein Graph mit n Knoten und m Kanten. Zur Speicherung von G wird folgender Platz benötigt:

- Adjazenz-Matrix: $O(n^2)$, geeignet für **dichte** Graphen
- Adjazenz-Liste: $O(n + m)$

Adjazenz-Matrizen unterstützen sehr gut Aufgaben wie: Falls Knoten u und v adjazent sind, tue ...

Adjazenz-Listen unterstützen sehr gut das Verfolgen von Kanten (für alle zu Knoten u inzidenten Kanten tue ...)

Hinzufügen und Löschen von Knoten werden nicht unterstützt. Es gibt voll dynamische Datenstrukturen ...

Durchsuchen von gerichteten Graphen

Aufgabe: Die Suche soll alle von einem gegebenen Startknoten s aus erreichbaren Knoten finden.

Sei D eine Datenstruktur zum Speichern von Kanten.

Algorithmus:

markiere alle Knoten als „unbesucht“
markiere den Startknoten s als „besucht“
füge alle aus s auslaufenden Kanten zu D hinzu
solange D nicht leer ist:
    entnehme eine Kante (u, v) aus D
    falls der Knoten v als „unbesucht“ markiert ist:
        markiere Knoten v als „besucht“
        füge alle aus v auslaufenden Kanten zu D hinzu

Durchsuchen von gerichteten Graphen (2)

Anmerkung: In der Datenstruktur D speichern wir diejenigen Kanten, von denen vielleicht noch unbesuchte Knoten erreicht werden können.

Laufzeit:

- jede Kante wird höchstens einmal in D eingefügt
- jeder Knoten wird höchstens einmal inspiziert

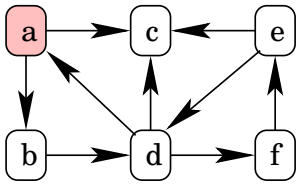
$\Rightarrow$ Laufzeit proportional zur Anzahl der vom Startknoten aus erreichbaren Knoten und Kanten, also $O(|V| + |E|)$

Typ der Datenstruktur bestimmt die **Durchlaufordnung:**

- Stack (last in, first out): **Tiefensuche**
- Liste (first in, first out): **Breitensuche**

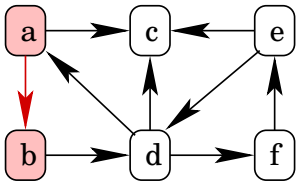
Tiefensuche

(1)



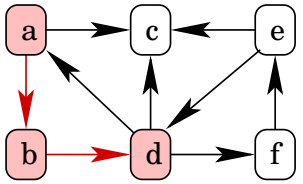
D = {(a,b), (a,c)}

(2)



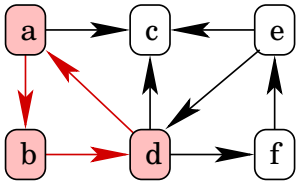
D = {(b,d), (a,c)}

(3)



D = {(d,a), (d,c), (d,f), (a,c)}

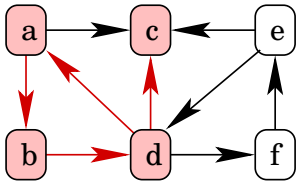
(4)



D = {(d,c), (d,f), (a,c)}

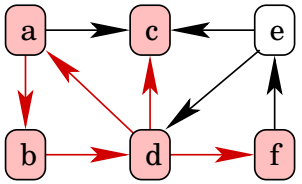
Tiefensuche (2)

(5)



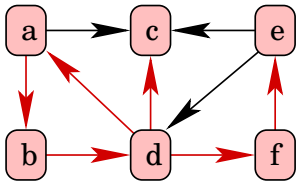
D = {(d,f), (a,c)}

(6)



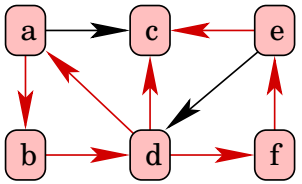
D = {(f,e), (a,c)}

(7)



D = {(e,c), (e,d), (a,c)}

(8)



D = {(e,d), (a,c)} . . .

Tiefensuche (3)

- **Baumkante:** Kante, der die Tiefensuche folgt
- **Vorwärtskante:** Kante $(u, v) \in E$ mit dfb[v] $>$ dfb[u], die keine Baumkante ist
- **Querkante:** Kante $(u, v) \in E$ mit dfb[v] $<$ dfb[u] und dfe[v] $<$ dfe[u]
- **Rückwärtskante:** Kante $(u, v) \in E$ mit dfb[v] $<$ dfb[u] und dfe[v] $>$ dfe[u]

Rekursive Tiefensuche:

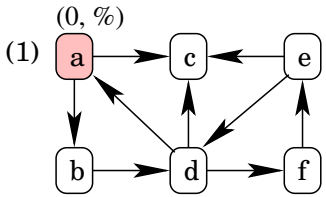
```
markiere alle Knoten als „unbesucht“
dfbZähler := 0
dfeZähler := 0
tiefensuche(s)
```

Tiefensuche (4)

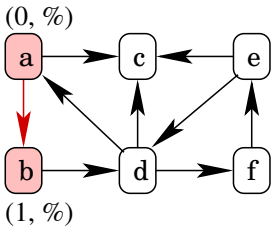
tiefensuche(u):

markiere u als „besucht“
dfb[u] := dfbZähler
dfbZähler := dfbZähler + 1
betrachte alle Kanten $(u, v) \in E$:
    falls Knoten v als „unbesucht“ markiert ist:
        tiefensuche(v)
dfe[u] := dfeZähler
dfeZähler := dfeZähler + 1

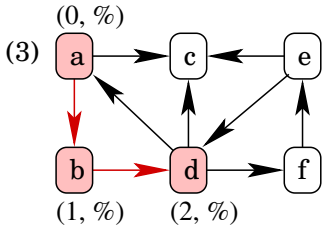
Tiefensuche (5)

(1)


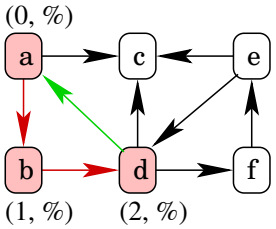
D = {(a,b), (a,c)}

(2)


D = {(b,d), (a,c)}

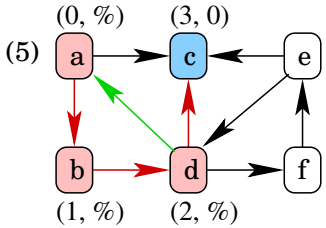
(3)


D = {(d,a), (d,c), (d,f), (a,c)}

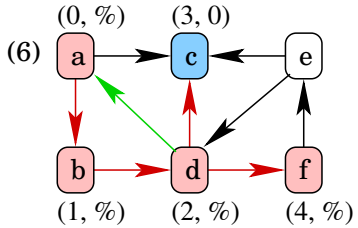
(4)


D = {(d,c), (d,f), (a,c)}

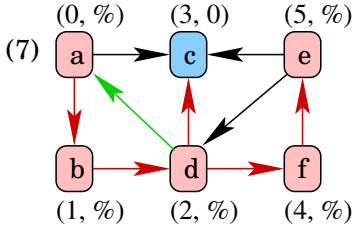
Tiefensuche (6)

(5)


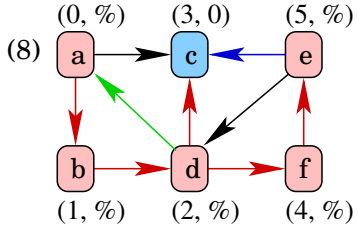
D = {(d,f), (a,c)}

(6)


D = {(f,e), (a,c)}

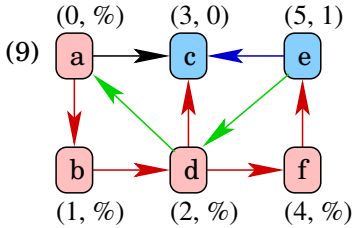
(7)


D = {(e,c), (e,d), (a,c)}

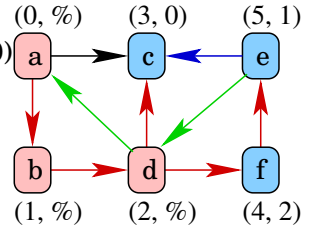
(8)


D = {(e,d), (a,c)}

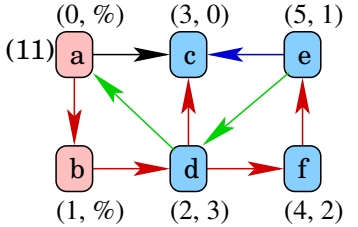
Tiefensuche (7)

(9)


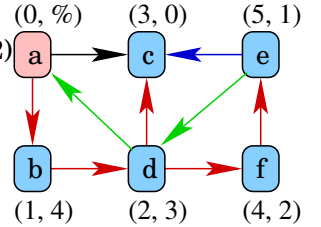
D = {(a,c)}

(10)


D = {(a,c)}

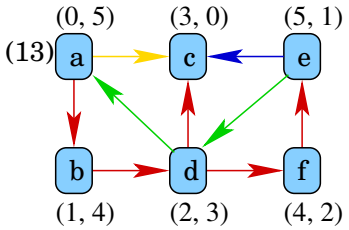
(11)


D = {(a,c)}

(12)


D = {(a,c)}

Tiefensuche (8)

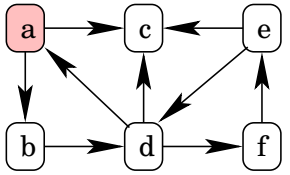


D = {}

- rot markierte Kanten: Baumkanten
- grün markierte Kanten: Rückwärtskanten
- blau markierte Kanten: Querkanten
- gelb markierte Kanten: Vorwärtskanten

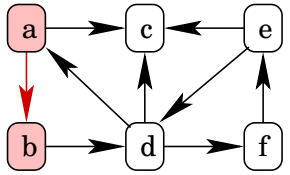
Breitensuche

(1)



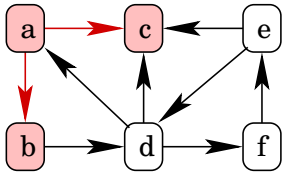
D = {(a,b), (a,c)}

(2)



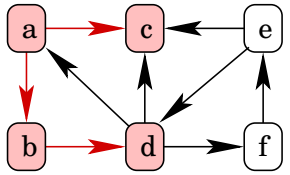
D = {(a,c), (b,d)}

(3)



D = {(b,d)}

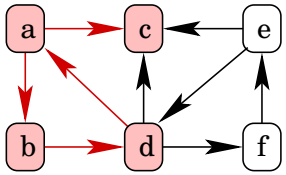
(4)



D = {(d,a), (d,c), (d,f)}

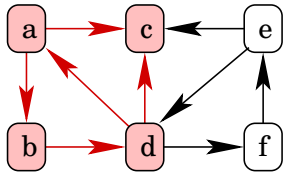
Breitensuche (2)

(5)



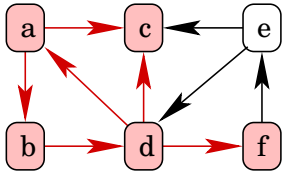
D = {(d,c), (d,f)}

(6)



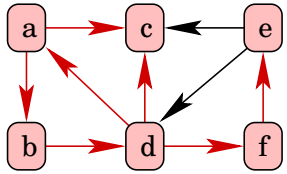
D = {(d,f)}

(7)



D = {(f,e)}

(8)



D = {(e,c), (e,d)} . . .

Durchsuchen von ungerichteten Graphen

Obige Suche funktioniert mit leichter Modifikation auch für ungerichtete Graphen.

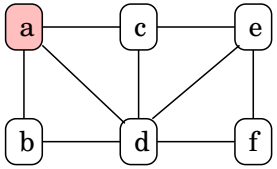
Sei D eine Datenstruktur zum Speichern von Kanten.

Algorithmus:

markiere alle Knoten als „unbesucht“
markiere den Startknoten s als „besucht“
füge alle mit s inzidenten Kanten zu D hinzu
solange D nicht leer ist:
    entnehme eine Kante $\{u, v\}$ aus D
    falls der Knoten u/v als „unbesucht“ markiert ist:
        markiere Knoten u/v als „besucht“
        füge alle zu u/v inzidenten Kanten zu D hinzu

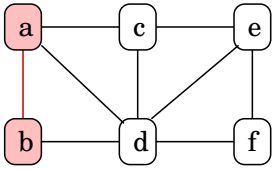
Breitensuche: ungerichtet

(1)



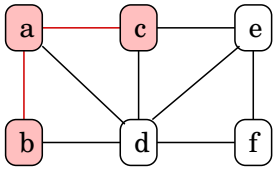
D = {{a,b}, {a,c}, {a,d}}

(2)



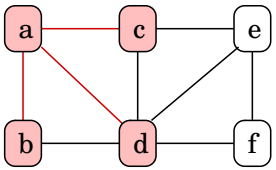
D = {{a,c}, {a,d}, {b,d}}

(3)



D = {{a,d}, {b,d}, {c,d}, {c,e}}

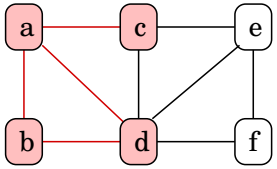
(4)



D = {{b,d}, {c,d}, {c,e}, {d,e}, {d,f}}

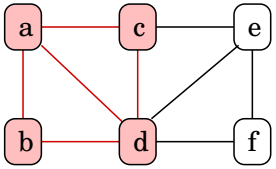
Breitensuche: ungerichtet (2)

(5)



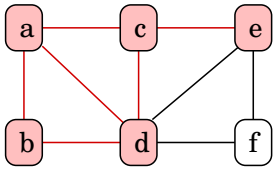
D = {{c,d}, {c,e}, {d,e}, {d,f}}

(6)



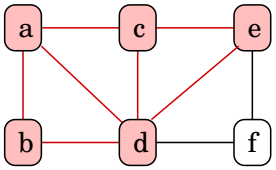
D = {{c,e}, {d,e}, {d,f}}

(7)



D = {{d,e}, {d,f}, {e,f}}

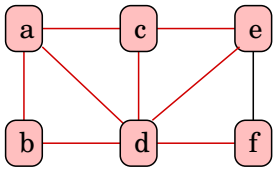
(8)



D = {{d,f}, {e,f}}

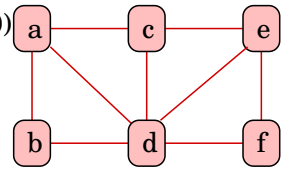
Breitensuche: ungerichtet (3)

(9)



D = {{e,f}}

(10)



D = {}

Minimale Spannbäume

Definition:

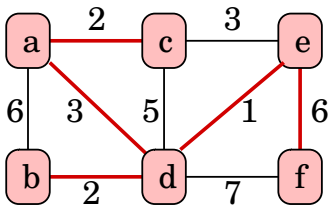
- ein ungerichteter Graph $G = (V, E)$ heißt **zusammenhängend**, wenn gilt: für alle $u, v \in V$ existiert ein Weg von u nach v.
- ein **Spannbaum** T von $G = (V, E)$ ist ein zusammenhängender Teilgraph $T = (V, E')$ von G mit $|V| - 1$ Kanten

gegeben: ein ungerichteter, zusammenhängender Graph $G = (V, E)$ mit Kostenfunktion $c : E \to \mathbb{R}^+$

gesucht: ein Spannbaum $T = (V, E')$ von G mit minimalen Kosten:

$$c(T) = \sum_{e \in E} c(e)$$

Minimale Spannbäume: Beispiel



Algorithmen:

- Kruskal: basiert auf Union-Find Datenstruktur, Laufzeit $O(|E| \cdot \log |V|)$
- Karger, Klein, Tarjan [1]: zur Zeit bester Algorithmus, randomisiert, erwartete Laufzeit $O(|V| + |E|)$

---

[1] A randomized linear-time algorithm to find minimum spanning trees. Journal of the ACM, 1995.

Minimale Spannbäume: Motivation

Verkabelung und Routing

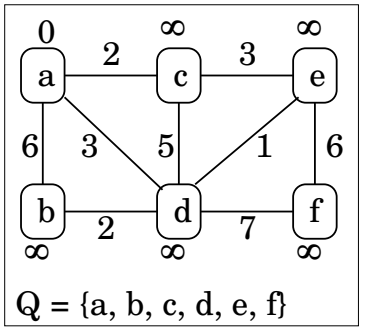
- alle Häuser ans Telefonnetz anschließen: aus Kostengründen möglichst wenig Kabel verlegen
- Stromversorgung von elektrischen Bauteilen auf einer Platine
- Routing:
  * CISCO IP Multicast
  * Spanning Tree Protocol
- es werden nur die Straßen repariert, so dass nach wie vor alle Häuser erreichbar sind

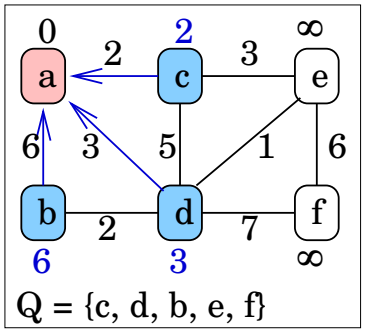
Prims Algorithmus

Sei Q eine Datenstruktur zum Speichern von Knoten. Die Knoten sind mit den Kantenwerten gewichtet.

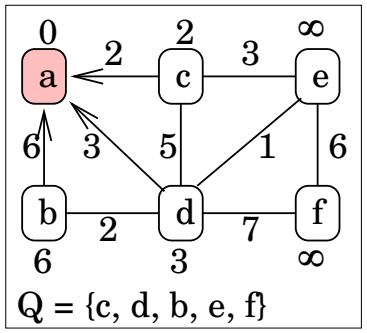
$Q := V$
$key[v] := \infty$ for all $v \in V$
$key[s] := 0$ for some arbitrary $s \in V$
while $Q \neq \emptyset$ do
    $u :=$ ExtractMin(Q)
    for all $v \in Adj(u)$ do
        if $v \in Q$ and $c((u,v)) < key[v]$
        then $key[v] := c((u,v))$
            $\pi[v] := u$

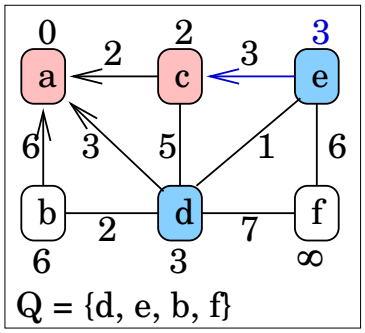
Prims Algorithmus: Beispiel (a)



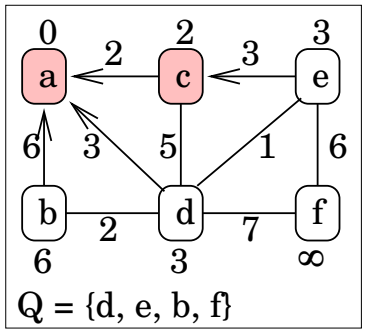


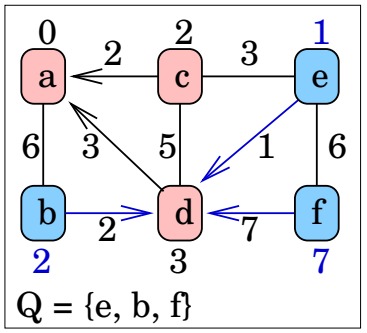
Prims Algorithmus: Beispiel (b)



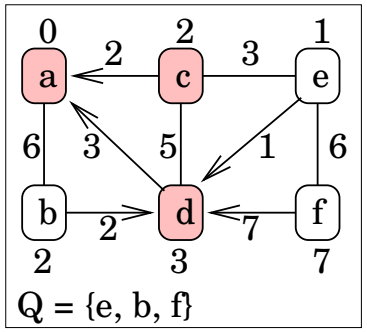


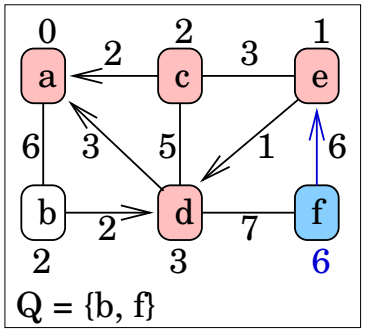
Prims Algorithmus: Beispiel (c)



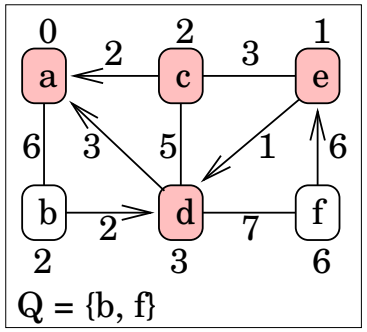


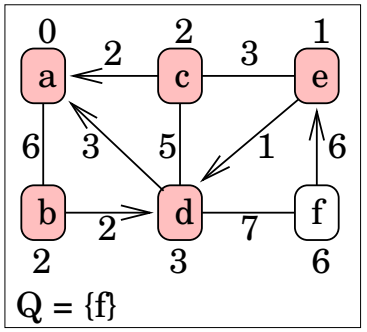
Prims Algorithmus: Beispiel (d)



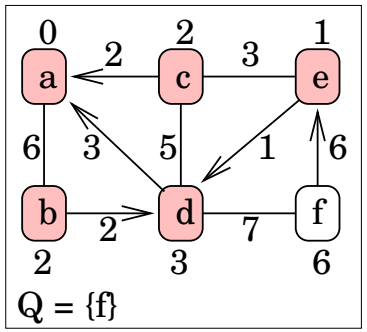


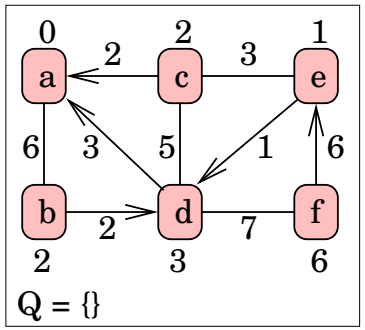
Prims Algorithmus: Beispiel (e)





Prims Algorithmus: Beispiel (f)





Prims Algorithmus: Bewertung

Laufzeit:

$$T = \Theta(V) \cdot T_{ExtractMin} + \Theta(E) \cdot T_{DecreaseKey}$$

Wird die Datenstruktur Q mittels Array implementiert:

- $T_{ExtractMin} \in O(V)$
- $T_{DecreaseKey} \in O(1)$

$\Rightarrow$ Laufzeit $O(V^2)$

Anmerkungen: implementiere Q als

- Binär-Heap: Laufzeit $O(E \log V)$
- Fibbonacci-Heap: Laufzeit $O(E + V \log V)$

Kürzeste Wege

gegeben: ein ungerichteter, zusammenhängender Graph $G = (V, E)$ mit Kostenfunktion $c : E \to \mathbb{R}^+$

single source shortest paths:
Suche von einem gegebenen Knoten $s \in V$ die kürzesten Wege zu allen anderen Knoten.

all pairs shortest paths:
Suche für jedes Paar $(u, v) \in V^2$ den kürzesten Weg von u nach v.

Wir betrachten hier nur die erste Variante.

Kürzeste Wege: Motivation

Reiseplanung: Finde den kürzesten Weg zum Urlaubsort.

Kostenminimierung: Finde Zugverbindung

- mit möglichst kurzer Reisezeit,
- bei der man möglichst wenig umsteigen muss, oder
- die möglichst preiswert ist.

$\Rightarrow$ z.B. Fahrplanberechnung der Deutschen Bahn AG

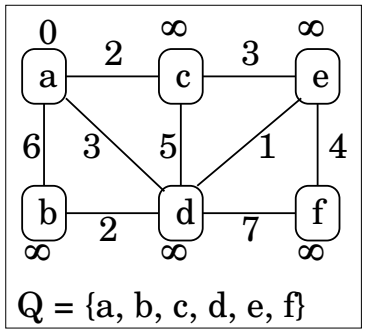
Routing im Internet: OSPF (Open Shortest Path First)

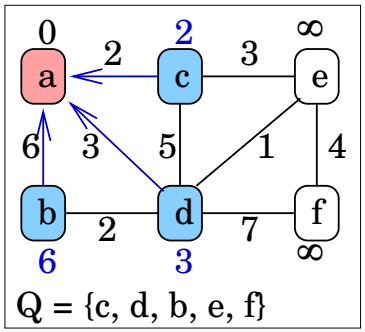
Dijkstras Algorithmus

Sei Q eine Datenstruktur zum Speichern von Knoten. Die Knoten sind mit den Kantenwerten gewichtet.

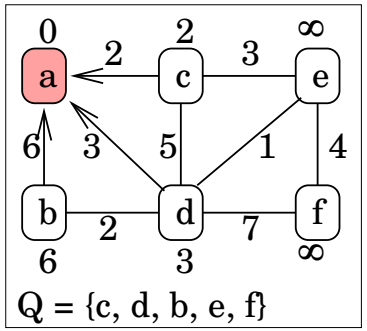
$Q := V$
$d[v] := \infty$ for all $v \in V$
$d[s] := 0$ for some arbitrary $s \in V$
while $Q \neq \emptyset$ do
    $u :=$ ExtractMin(Q)
    for all $v \in Adj(u)$ do
        if $v \in Q$ and $d[v] > d[u] + c((u,v))$
        then $d[v] := d[u] + c((u,v))$
            $\pi[v] := u$

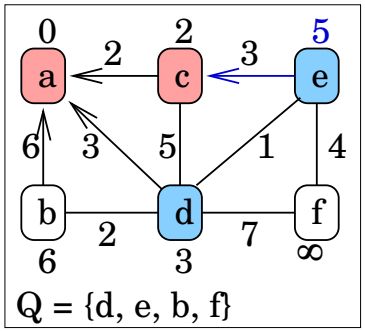
Dijkstras Algorithmus: Beispiel (a)



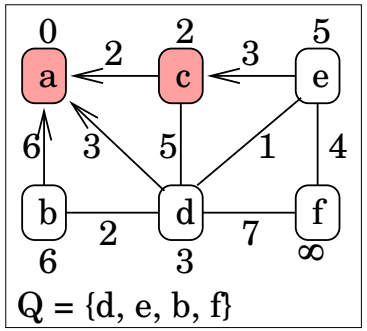


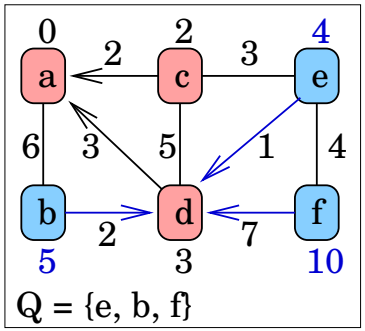
Dijkstras Algorithmus: Beispiel (b)



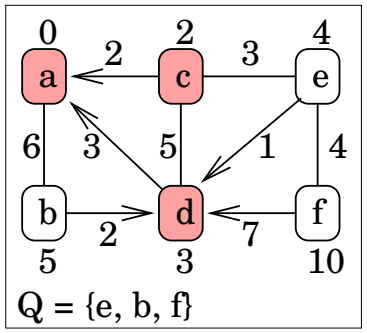


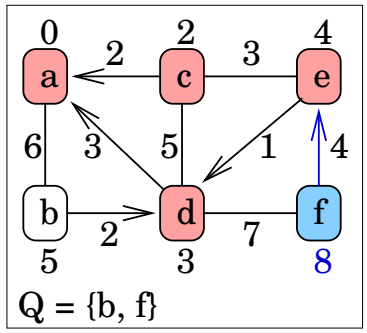
Dijkstras Algorithmus: Beispiel (c)



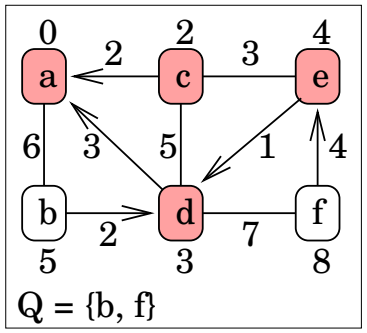


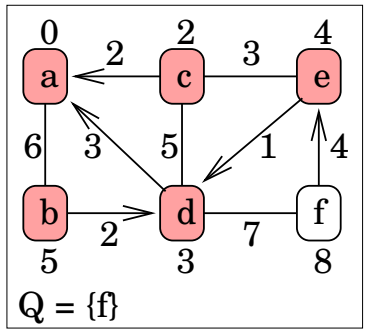
Dijkstras Algorithmus: Beispiel (d)



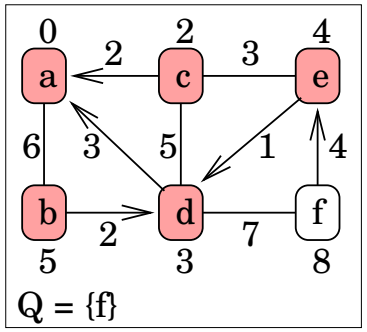


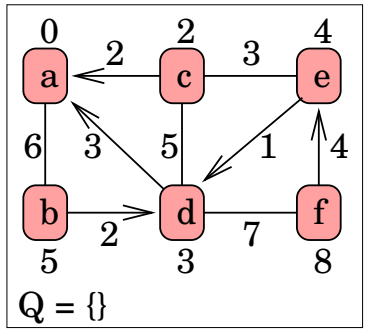
Dijkstras Algorithmus: Beispiel (e)





Dijkstras Algorithmus: Beispiel (f)





Dijkstras Algorithmus: Bewertung

Laufzeit:

$$T = \Theta(V) \cdot T_{ExtractMin} + \Theta(E) \cdot T_{DecreaseKey}$$

Gleiche Laufzeit wie bei Prims Algorithmus!
Implementiere Q als

- Array: Laufzeit $O(V^2)$
- Binär-Heap: Laufzeit $O(E \log V)$
- Fibbonacci-Heap: Laufzeit $O(E + V \log V)$

Ungewichtete Graphen: modifizierte Breitensuche

Algorithmen & Datenstrukturen

Datenstrukturen: Einführung

Die richtige Organisationsform einer Menge von Daten hat erheblichen Einfluss darauf, wie effizient sich bestimmte Operationen ausführen lassen! → Prims Algorithmus

Aufgabe: Finde Speicherungsform, so dass die Operationen möglichst effizient ausführbar sind.

Wesentliche Einflussgrößen:

- Effizienz bzgl. Laufzeit, Speicherbedarf oder einfache Programmierbarkeit???
- Ist die Folge der Operationen bekannt? oder:
  Sind relative Häufigkeiten der Operationen bekannt?

Datenstrukturen: Einführung (2)

Gesucht: Datenstruktur für einfügen, suchen, entfernen

Implementierung 1: unsortiertes Array

- einfügen: am Ende anhängen → $O(1)$
- suchen: lineare Suche → $O(n)$
- entfernen: finden und tauschen → $O(n)$

Implementierung 2: sortiertes Array

- einfügen: Position finden und verschieben → $O(n)$
- suchen: binäre Suche → $O(\log(n))$
- entfernen: finden und verschieben → $O(n)$

Datenstrukturen: Einführung (3)

Einfluss der relativen Häufigkeiten der Operationen auf die Laufzeit bei unterschiedlichen Implementierungen:
(Annahme: Belegung sei m, m viel größer als n)

- einfügen/suchen = 1/n

  unsortiert: $1 \cdot O(1) + n \cdot O(m) \rightarrow O(n \cdot m)$
  sortiert: $1 \cdot O(m) + n \cdot O(\log(m)) \rightarrow O(n \log(m))$

- einfügen/suchen = 1/1

  unsortiert: $1 \cdot O(1) + 1 \cdot O(m) \rightarrow O(m)$
  sortiert: $1 \cdot O(m) + 1 \cdot O(\log(m)) \rightarrow O(m)$

- einfügen/suchen = n/1

  unsortiert: $n \cdot O(1) + 1 \cdot O(m) \rightarrow O(m)$
  sortiert: $n \cdot O(m) + 1 \cdot O(\log(m)) \rightarrow O(n \cdot m)$

Datenstrukturen: Einführung (4)

Ein **abstrakter Datentyp** besteht aus einer oder mehreren Mengen von Objekten und darauf definierten Operationen.

Beispiel: Polynom

- Objektmenge: Polynome mit ganzzahligen Koeffizienten
- Operationen: Addition, Multiplikation, erste Ableitung

Beispiel: Landkarte

- Objektmenge: Städte, Wege zwischen den Städten
- Operationen: nächster Nachbar, Distanz zweier Städte

Daten und Operationen auf den Daten gehören zusammen! → modulare Programmierung

Datenstrukturen: Einführung (5)

Warum **abstrakter** Datentyp? Die Semantik (Bedeutung) der Operationen kann unabhängig von der Realisierung axiomatisch definiert werden.

Beispiel: Definition eines Stack (last in first out)

- empty(init()) = true
- empty(push(s,e)) = false
- pop(push(s,e)) = s
- pop(init()) = undefiniert
- top(push(s,e)) = e
- top(init()) = undefiniert

→ Mathematischer Hintergrund: Algebra

Datenstrukturen: Einführung (6)

Die Axiome definieren das Zusammenspiel der Daten, nicht die Implementierung der Operationen.

Beispiel: Kalenderdatum

- differenz(d, addiere(d,n)) = n
- wochentag(d) = wochentag(addiere(d,7))

Nur Kenntnis der Axiome notwendig, um den Datentyp anwenden zu können. Implementierung kann also jederzeit geändert werden, muss aber die Axiome garantieren.

- C/C++: Header-Datei
- Java: Interface

Schnittstellendefinition wichtig für große Projekte!

Datenstrukturen: Einführung (7)

Unterscheidung:

- **Datentyp:** die vorhandenen Grundtypen (`int`, `float`, ...) und die daraus mittels Strukturierungsmethoden (`union`, `struct`, ...) gebildeten zusammengesetzten Typen.
- **abstrakter Datentyp:** besteht aus einer oder mehreren, mit mathematischen Methoden festgelegten Mengen von Objekten und darauf definierten Operationen.
- **Datenstruktur:** konkrete Implementierung der Objektmengen eines abstrakten Datentyps.

Datenstruktur: Baum

- in der Informatik weit verbreitet: Entscheidungsbäume, Syntaxbäume, Ableitungsbäume, ...
- gehören zu den wichtigsten Datenstrukturen:
  * spannende Bäume
  * Suchbäume
  * Verzeichnisbäume (hierarchische Dateisysteme)
  * hierarchische Datenbanksysteme (LDAP)
- verallgemeinerte Liste: ein Element (**Knoten**) hat nicht nur einen Nachfolger (lineare Liste), sondern eine endliche, begrenzte Anzahl Nachfolger (**Kinder**)

Datenstruktur: Baum (2)

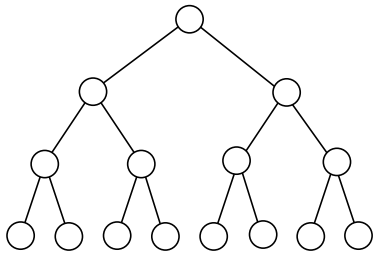
Begriffe:

- der Knoten ohne Vorgänger (**Elter**) heißt **Wurzel**
- Knoten ohne Nachfolger heißen **Blätter**
- **innere Knoten:** Knoten $\neq$ Blätter
- **Binärbaum:** Baum, bei dem jeder innere Knoten genau zwei Kinder hat (allgemein: k-näre Bäume)
- **Tiefe eines Knotens:** Abstand des Knotens von der Wurzel (Anzahl der Kanten)
- **Höhe des Baumes:** maximale Abstand eines Blattes von der Wurzel (Anzahl der Kanten)

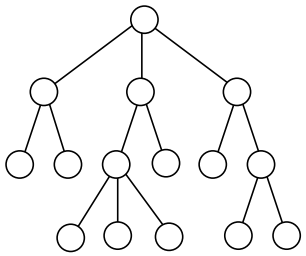
Datenstruktur: Baum (3)

Begriffe: (Fortsetzung)

- auf den Nachfolgern kann eine **Ordnung** bestehen: linker/rechter Nachfolger, i-ter Nachfolger, ...
- **vollständiger Baum**: in jeder Ebene maximale Anzahl Knoten und alle Blätter haben gleiche Tiefe



vollständiger Binärbaum

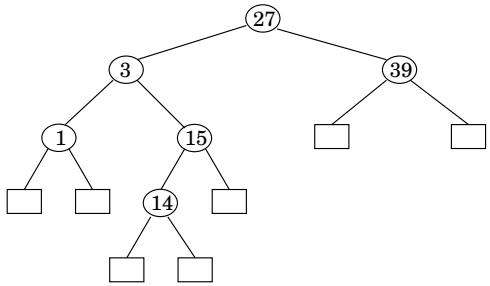


ternärer Baum

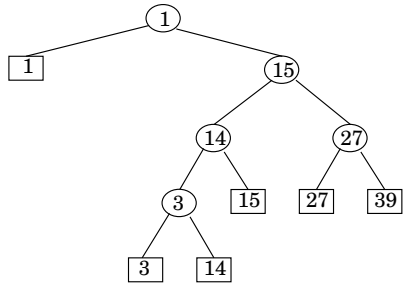
Suchbäume

Suchbäume sind geordnete binäre Bäume. Unterscheidung:

- **Suchbäume:** innere Knoten speichern Schlüsselwerte
- **Blattsuchbäume:** Blätter speichern Schlüsselwerte, innere Knoten nur Wegweiser (max. Wert des linken Teilbaums)



Suchbaum



Blattsuchbaum

Suchbäume (2)

Suche eines Schlüssels k in einem Suchbaum:

1. beginne bei der Wurzel p
2. vergleiche k mit dem bei p gespeicherten Schlüssel k_p
   - $k < k_p$: setze Suche mit linkem Nachfolger von p fort
   - $k > k_p$: setze Suche mit dem rechten Kind von p fort

$\Rightarrow$ wird ein Blatt erreicht, ist k nicht im Baum gespeichert

Einfügen eines Schlüssels k in einen Suchbaum:

1. suche den Schlüssel k im Suchbaum
2. falls k nicht im Baum enthalten ist, endet die Suche bei einem Blatt: füge k ein und erzeuge zwei neue Blätter

Suchbäume (3)

Problem: Die Form des Suchbaums (also die Laufzeit der Suchoperation) hängt stark von der Einfügereihenfolge ab!

Lösung: AVL-Bäume, Rot/Schwarz-Bäume, B/B*-Bäume

Implementierung der Suchbäume: Übungsaufgabe

Suchbäume (4)

Entfernen eines Schlüssels k aus einem Suchbaum:

1. suche den Schlüssel k im Suchbaum (Knoten sei u)
2. unterscheide anhand der Nachfolger von u:
   - beide Nachfolger sind Blätter: mache u zu einem Blatt
   - nur ein Nachfolger ist Blatt: hänge inneren Knoten an den Elter von u
   - beide Nachfolger sind innere Knoten: suche im rechten Teilbaum von u den kleinsten Schlüssel $k' > k$. Der Knoten v, der k' speichert, heißt symmetrischer Nachfolger von u. Ersetze k durch k' und entferne Knoten v

Formale Sprachen

Formale Sprachbeschreibung

Lexikalische Regeln definieren die Wörter, aus denen Programme aufgebaut werden dürfen → Vokabeln

Syntaktische Regeln legen fest, wie aus den Wörtern korrekte Programme gebildet werden → Satzbau

- Die Überprüfung erfolgt mit einem **Parser**.
- Die Syntax wird durch eine **Grammatik** definiert.

Nicht alle Spracheigenschaften können syntaktisch festgelegt werden → **semantische Regeln:**

- Wurde Bezeichner vor seiner Benutzung deklariert?
- Sind keine Typfehler vorhanden?

Grammatik

Sei Σ eine endliche Menge von **Symbolen** (Buchstaben). Dann ist die **Menge alle Wörter** über Σ definiert als

$$\Sigma^* = \{a_1 a_2 \dots a_n \mid n \geq 0,\ a_i \in \Sigma\}.$$

Beispiel: Sei $\Sigma = \{a, b, c\}$, dann ist

$$\Sigma^* = \{\epsilon, a, b, c, aa, ab, ac, ba, bb, bc, ca, cb, cc, aaa, \dots\}.$$

- Die Elemente in Σ^* heißen **Wörter**.
- Wir schreiben die Buchstaben der Wörter direkt nebeneinander, nicht durch Kommata getrennt.
- Das leere Wort bezeichnen wir mit ϵ.
- Iterationsoperator $*$ wird auch **Kleene-Star**-Operator genannt (nach S.C. Kleene).

Grammatik (2)

Eine **kontextfreie Grammatik** ist ein 4-Tupel (N, Σ, P, S), wobei gilt:

1. N: endliche Menge **nicht-terminaler Symbole**
2. Σ: endliche Menge **terminaler Symbole**
3. $P \subseteq N \times (\Sigma \cup N)^*$: endliche Menge von **Produktionen**
4. $S \in N$: das **Startsymbol**

Schreibweise:

- schreibe nicht-terminale Symbole groß (A, B, C)
- schreibe terminale Symbole klein (a, b, c)
- Produktionen: $A \to w$ anstelle von (A, w)
- $A \to u \mid v \mid \dots$ ist Kurzschreibweise zu $(A, u), (A, v), \dots$

Grammatik (3)

Sei $G = (N, \Sigma, P, E)$ eine kontextfreie Grammatik mit $N = \{E, I, D\}$, $\Sigma = \{(,), 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, +, -, *, /\}$ und

$$P = \left\{ \begin{array}{lcl} E & \to & I \mid (E) \mid E + E \mid E - E \mid E * E \mid E/E \\ I & \to & 0 \mid 1D \mid 2D \mid 3D \mid \dots \mid 8D \mid 9D \\ D & \to & 0D \mid 1D \mid 2D \mid 3D \mid \dots \mid 8D \mid 9D \mid \epsilon \end{array} \right\}$$

Die Sprache $L(G)$ der Grammatik G ist die Menge aller Wörter $w \in \Sigma^*$, die aus dem Startsymbol ableitbar sind. Hier: arithmetische Ausdrücke.

Beispiel: Ableitung von $2 * (8 + 7)$

$$\begin{array}{rcl} E & \to & E * \underline{E} \to E * (\underline{E}) \to \underline{E} * (E + E) \to \underline{I} * (E + E) \\ & \to & 2\underline{D} * (E + E) \to 2 * (\underline{E} + E) \to 2 * (\underline{I} + E) \\ & \to & 2 * (8\underline{D} + E) \to 2 * (8 + \underline{E}) \to \dots \to 2 * (8 + 7) \end{array}$$

Grammatik (4)

Anmerkungen:

- Für eine Produktion $A \to u$ ist A die **linke Seite** und u die **rechte Seite der Produktion**.
- kontextfreie Grammatik: linke Seite der Produktion besteht aus genau einem nicht-terminalen Symbol $\to$ die Ersetzung der linken Seite erfolgt unabhängig von der Umgebung.
- Es gibt auch **kontext-sensitive Grammatiken**.

Grammatik (5)

Beispiel: Kontext-sensitive Grammatik

Sei $G = (N, \Sigma, P, S)$ mit $N = \{A, B, S\}$, $\Sigma = \{a, b\}$ und

$$P = \left\{ \begin{array}{rcl} S & \to & ABS \mid \epsilon \\ A & \to & a \\ B & \to & b \\ AB & \to & BA \\ BA & \to & AB \end{array} \right\}$$

Dann ist $L(G) = \{w \in \{a, b\}^* \mid \#a = \#b\}$.

Backus Naur Form

Die BNF-Notation wurde von John Backus und Peter Naur für die Definition von Algol 60 entwickelt.

BNF-Notation wurde später zur **Extended BNF** (EBNF) Notation erweitert.

Programmiersprachen: oft in EBNF-Notation beschrieben

In der EBNF-Notation werden Sprachen über kontextfreie Grammatiken (N, Σ, P, S) definiert, wobei die Produktionsmenge durch Regeln der Art $A ::= R$ beschrieben ist.

Extended Backus Naur Form

Form einer Regel:

- linke Seite: nicht-terminales Symbol A
- rechte Seite: ein **EBNF-Ausdruck** R über $\Sigma \cup N$ mit:
  - spitze Klammern < und >: nicht-terminale Symbole
  - der senkrechte Strich | kennzeichnet Alternativen
  - geschweifte Klammern { und }: Kleene-Star-Operator
  - eckige Klammern [und] bedeuten entweder null oder einmal
  - runde Klammern (und) dienen zur Klammerung

Extended Backus Naur Form (2)

Beispiel: ganzzahlige arithmetische Ausdrücke

```
<exp> ::= <int> | <exp> (+|-|*|/) <exp>
<int> ::= <dig> | <dig><int>
<dig> ::= 0|1|2|3|4|5|6|7|8|9
```

Anmerkungen:

- Um die in EBNF-Ausdrücken verwendeten Metasymbole von terminalen Symbolen zu unterscheiden, werden sie im Konfliktfall fett geschrieben.
- Gelegentlich erfolgt die Kennzeichnung der terminalen Symbole durch Anführungszeichen.

Extended Backus Naur Form (3)

Eine EBNF-Beschreibung für Bezeichner in C:

```
<name> ::= <char> {<dig> | <char>}
<char> ::= A|B|C|...|Z|a|b|c|...|z|_
<dig>  ::= 0|1|2|3|4|5|6|7|8|9
```

und dasselbe rekursiv:

```
<name> ::= <char> | <name><char> | <name><dig>
<char> ::= A|B|C|...|Z|a|b|c|...|z|_
<dig>  ::= 0|1|2|3|4|5|6|7|8|9
```

Alle mittels EBNF definierbare Sprachen sind kontextfreie Sprachen.

Syntaxdiagramme

Die Syntax einer Programmiersprache lässt sich grafisch mit Syntaxdiagrammen darstellen:

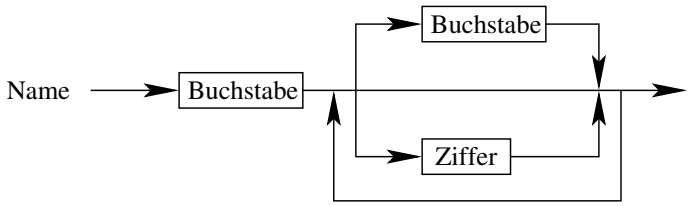
- nicht-terminale Symbole in rechteckigen Kästen
- terminale Wörter in runden Kästen
- Pfeile kennzeichnen möglichen weiteren Verlauf

Syntaktisch korrekter Text: durchlaufe das Syntaxdiagramm vom Eingangspfeil zum Ausgangspfeil und notiere dabei alle Wörter in runden Kästen, auf die man trifft.

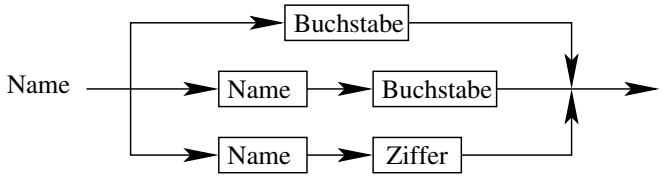
Syntaxdiagramme können auch **rekursiv** sein, d.h. im Diagramm mit Bezeichner X (bzw. in einem von X indirekt angegebenen Diagramm) kommt X selbst wieder vor.

Syntaxdiagramme (2)

Beispiel: Syntaxdiagramm für Bezeichner in C



und dasselbe rekursiv



Syntaxdiagramme (3)

Durch Syntaxdiagramme definierte Sprachen sind kontextfreie Sprachen. Syntaxdiagramme haben dieselbe Beschreibungskraft wie EBNF-Ausdrücke.

ergänzende Literatur:
J.E. Hopcroft, J.D. Ullman: Einführung in die Automatentheorie, Formale Sprachen und Komplexitätstheorie. Addison-Wesley.

Programmiersprachen

Programmiersprachen

Maschinenbefehle: elementare Operationen, die der Prozessor des Rechners unmittelbar ausführen kann.

- Daten aus dem Speicher lesen
- elementare arithmetische Operationen
- Daten in den Speicher schreiben
- Berechnung an anderer Stelle fortsetzen (Sprünge)

moderne Programmiersprachen: orientieren sich am zu lösenden Problem.

- abstrakte Formulierung des Lösungsweges
- Eigenheiten der Hardware werden nicht berücksichtigt

Programmiersprachen (2)

Konzepte:

- Werte und Typen
- Variablen und Befehle
- Bindungen
- Abstraktion
- Kapselung
- Typsysteme
- Ablaufsteuerung mit Ausnahmen
- Nebenläufigkeit

Programmiersprachen (3)

Werte und Typen:

- Daten sind genau so wichtig wie Programme: Telefonbuch, Wörterbuch, Satellitendaten, usw.
- **Wert:** beliebige Größe, die während einer Berechnung existiert
- **Typen:** Mengen von Werten, die in der Programmiersprache als Daten manipuliert werden können
  * einfache oder zusammengesetzte Typen
  * rekursive Typen (enthalten Werte desselben Typs)
- **Typsysteme:** schränken die Operationen ein, statische oder dynamische Typisierung
- **Ausdrücke:** berechne aus alten Werten neue Werte

Programmiersprachen (4)

Variablen und Befehle:

- **Variable:** Objekt, das einen Wert enthält
  * modelliert Objekte der realen Welt
  * wird durch Zuweisung überschrieben
- **Speicher:** Zusammenfassung von Zellen, besitzt einen gegenwärtigen Inhalt (zeitabhängig)
- **Lebensdauer:** lokale, globale, Heap- und persistente Variablen
- **Befehle:** Zuweisungen, Prozeduraufrufe, bedingte Befehle, sequentielle oder nebenläufige Blöcke, Iteration (Wiederholung, Schleife)
- **Seiteneffekte:** Auswerten eines Ausdrucks soll einen Wert liefern, aber sonst keinen weiteren Effekt haben

Programmiersprachen (5)

Bindungen:

- Bezeichner an Konstanten, Variablen, Prozeduren und Typen binden
- unterscheide Programmiersprachen: welche Arten von Größen können an Bezeichner gebunden werden?
- **Reichweite:** Teil des Programmtextes, für den die Vereinbarung gilt → Blockstruktur
  * **statische Reichweite:** zur Übersetzungszeit bekannt
  * **dynamische Reichweite:** erst zur Ausführungszeit bestimmt
- **Sichtbarkeit:** Bezeichner in verschiedenen Blöcken vereinbaren → in der Regel wird in jedem Block eine andere Größe bezeichnet

Programmiersprachen (6)

Abstraktion:

- Konstrukte der Programmiersprache sind Abstraktion von Maschinenbefehlen
- unterscheide zwei Fragestellungen:
  * **Was** tut ein Programmstück? → Prozedur aufrufen
  * **Wie** ist es implementiert? → Prozedur schreiben
- Hierarchiestufen:
  * baue einfache Prozeduren aus Befehlen auf
  * baue komplexe Prozeduren aus einfacheren auf
  * baue sehr komplexe Prozeduren aus komplexen auf ...
- Abstraktionen: Prozeduren und Funktionen

Programmiersprachen (7)

Kapselung: → Programmieren im Großen

- Setze große Programme aus Modulen zusammen!
  analog: Fernseher/Computer besteht aus Baugruppen
- **Modul:**
  * benannte Programmeinheit, die (mehr oder weniger) unabhängig vom Rest implementiert werden kann.
    Beispiele: Liste, Wörterbuch, ...
  * hat klar umrissenen Zweck und klare Schnittstelle zu anderen Modulen

  ⇒ **Wiederverwendbarkeit**
- nur wenige der Modul-Komponenten sind nach außen sichtbar → Abstraktion: Was tut das Modul, nicht wie!

Programmiersprachen (8)

in der Vorlesung *Programmentwicklung:*

- **Typsysteme:**
  * monomorph: jede Konstante, Variable, Funktion usw. muss von einem bestimmten Typ vereinbart werden → nicht ausreichend
  * deshalb: Überladen, Polymorphie, Vererbung
- **Ablaufsteuerung mit Ausnahmen**
- **Nebenläufigkeit:** bspw. bei GUI-Programmierung

Programmiersprachen (9)

Paradigmen:

- imperatives Programmieren
- objekt-orientiertes Programmieren
- funktionales Programmieren
- logisches Programmieren

Programmiersprachen (10)

imperatives Programmieren:

- beruht auf Befehlen, die Variablen im Speicher überschreiben (lat. *imperare*: befehlen)
- seit den 50er Jahren: Variablen und Zuweisungen sind nützliche Abstraktion von Lade- und Speicherbefehlen in Maschinensprachen → Basic, Cobol, Fortran
- heute extrem weit verbeitet (OOP ist Spezialfall der imperativen Programmierung) → C/C++, Delphi, Java
- natürliche Art der Modellierung von Prozessen der realen Welt: Zustand von Objekten der realen Welt ändert sich mit der Zeit → beschreiben durch Variablen

Programmiersprachen (11)

objekt-orientiertes Programmieren:

- Ein Modul, das globale Variablen benutzt, kann nicht unabhängig von anderen Modulen, die die Variable auch benutzen, entwickelt und verstanden werden!
- Schnittstellen: jede globale Variable wird in einem Modul gekapselt und mit einem Satz von Prozeduren versehen, die als einzige direkten Zugriff auf die Variable haben
- heutzutage nennt man solche Schnittstellen **Klassen**
- Klassen geben Programmen eine modulare Struktur
- man kann auch in C objekt-orientiert programmieren, aber es wird nicht erzwungen oder unterstützt

Programmiersprachen (12)

funktionales Programmieren:

- Programm als Implementierung einer Abbildung: Eingabewerte auf Ausgabewerte abbilden
- Konzepte:
  * **Mustervergleich:** ein Funktionsname für verschiedene Parametertypen
  * **Funktionen höherer Ordnung:** Parameter oder Ergebnis sind Funktionen
  * **verzögerte Auswertung:** das Argument einer Funktion wird erst bei der ersten Benutzung ausgewertet, nicht beim Aufruf der Funktion

Programmiersprachen (13)

logisches Programmieren:

- Programm berechnet Relation $\rightarrow$ allgemeiner als Abbildung, höhere Stufe als funktionales Programmieren
- Sei $R : S \times T$ eine zweistellige Relation:
  * gegeben a, b: bestimme, ob $R(a, b)$ gilt
  * gegeben a: finde alle $t \in T$, so dass $R(a, t)$ gilt
  * gegeben b: finde alle $s \in S$, so dass $R(s, b)$ gilt
  * finde alle $s \in S$ und $t \in T$, so dass $R(s, t)$ gilt
- volle Mächtigkeit der Prädikatenlogik kann nicht ausgeschöpft werden $\rightarrow$ beschränken auf Hornklauseln

Programmiersprachen (14)

ergänzende Literatur:

- Watt: Programmiersprachen. Carl Hanser Verlag
- Ghezzi, Jazayeri: Konzepte der Programmiersprachen. Oldenbourg Verlag
- Louden: Programming Languages. PWS Publishing

Programmiersprachen (15)

Programme: Text nach genau festgelegten Regeln, durch **Grammatik der Programmiersprache** definiert.

Grammatik-Regeln sind exakt einzuhalten, sonst wird das Programm als Ganzes nicht verstanden!

Beispiel:

```
#include <stdio.h>
main() {
    printf("Hello, world!\n")
}
```

übersetzen mit GNU C-Compiler liefert:

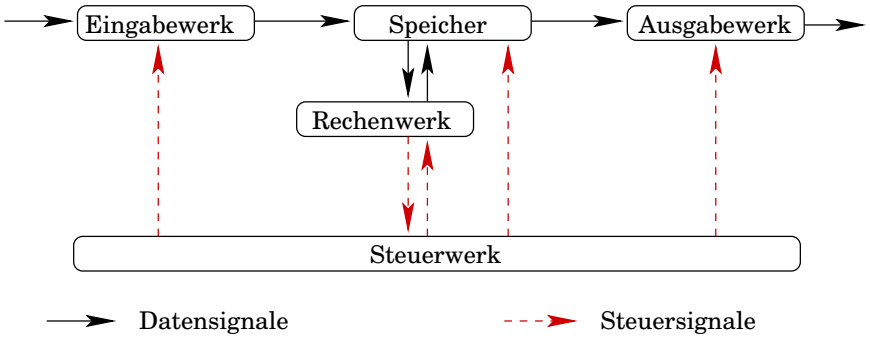
```
hw.c: In Funktion >>main<<:
hw.c:4: error: Fehler beim Parsen before '}' token
```


Rechnerarchitektur

Von-Neumann-Rechner

John von Neumann: 1903 - 1957, Mathematiker, schuf die wesentlichen theoretischen Grundlagen für programmgesteuerte Automaten → Basis heutiger Computer

Der Rechner besteht aus fünf **Funktionseinheiten**:



Von-Neumann-Rechner (2)

Ohne Programm ist die Maschine nicht arbeitsfähig: Zur Lösung eines Problems muss von außen ein Programm eingegeben und im Speicher abgelegt werden.

Programme, Daten, Zwischen- und Endergebnisse werden in demselben Speicher abgelegt.

der Speicher:

- unterteilt in gleichgroße Zellen
- Zellen sind fortlaufend nummeriert
- über die Nummer (Adresse) einer Speicherzelle kann deren Inhalt abgerufen oder verändert werden

Von-Neumann-Rechner (3)

Aufeinanderfolgende Befehle eines Programms werden in aufeinanderfolgenden Speicherzellen abgelegt.

- nächster Befehl: Steuerwerk $\rightarrow$ Befehlsadresse + 1
- **Sprungbefehle:** Abweichen von der Bearbeitung der Befehle in der gespeicherten Reihenfolge.

unterschiedliche Befehlsarten:

- Arithmetik: Addieren, Multiplizieren, Konstanten laden
- Logik: Vergleiche, logisches NICHT, UND, ODER
- Transport: Speicher zum Rechenwerk, Ein-/Ausgabe
- bedingte Sprünge
- sonstiges: Schieben, Unterbrechen, Warten

Von-Neumann-Rechner (4)

Das Rechenwerk besteht aus zwei Komponenten:

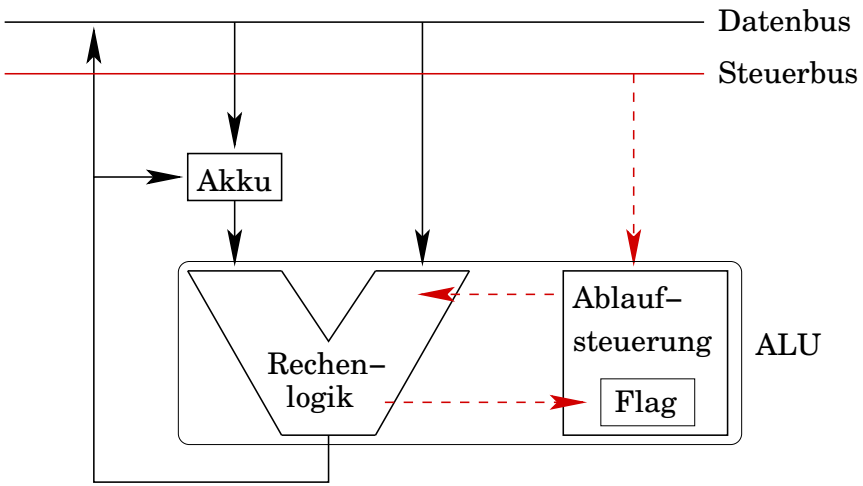
- **Akkumulator:** einfaches Register, ist als Operand an jeder Berechnung beteiligt und nimmt das Ergebnis auf.
- **ALU:** Arithmetic Logical Unit

Auch die **ALU** besteht aus zwei Komponenten:

- **Rechenlogik:** realisiert mathematische Funktionen
- **Ablaufsteuerung:**
  * erzeugt die sogenannten Flags
  * wählt die gewünschte Funktion aus
  * stellt komplexe Funktionalität bereit (Multiplikation durch Addition mittels Barrel-Shifter-Verfahren)

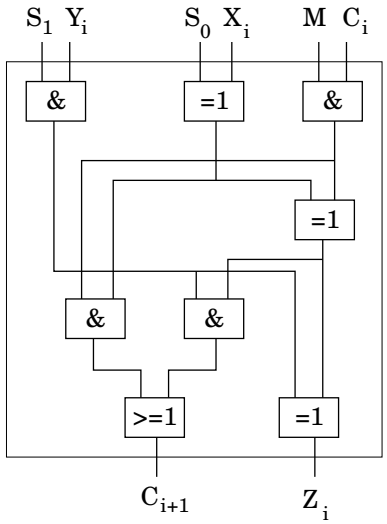
Von-Neumann-Rechner (5)

das Rechenwerk:



Von-Neumann-Rechner (6)

1-Bit ALU:



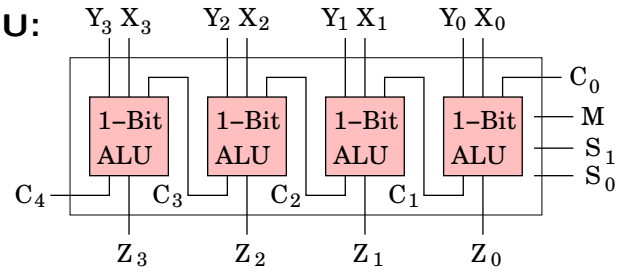
S_0, S_1, M: Steuerleitungen
$M = 0$: logischer Modus
$M = 1$: arithmetischer Modus

X_i, Y_i: Eingabewerte
C_i: Carry-Eingang

Z_i: Ergebnis
C_{i+1}: Carry-Bit

Von-Neumann-Rechner (7)

4-Bit ALU:



Eingabe-/Ausgabewerk: Logik, die den eigentlichen von-Neumann-Computer mit der *Außenwelt* verbindet:

- Grafikkarte
- PCI-Bus für Einsteckkarten
- SCSI-Interface zum Anschluß von Peripheriegeräten
- parallele (Drucker) und serielle (Maus) Schnittstellen

Von-Neumann-Rechner (8)

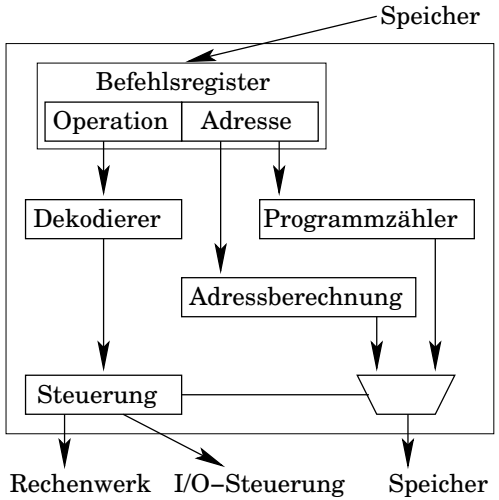
Alle Daten (Befehle, Adressen usw.) werden binär codiert. Geeignete Schaltwerke im Steuerwerk und an anderen Stellen sorgen für die richtige Entschlüsselung (Decodierung).

das Steuerwerk: Abarbeiten eines Befehls in drei Phasen

- **Laden:** holen des Befehls aus dem Speicher, dekodierern des Befehlscodes, ggf. berechnen einer Adresse
- **Verarbeiten:** ausführen der Anweisung, z.B. Addition
- **Speichern:** Ergebnis vom Akkumulator/Register in eine Speicherzelle schreiben

Von-Neumann-Rechner (9)

das Steuerwerk:



Steuerung: als endlicher Automat oder als Mikroprogramm realisiert

Von-Neumann-Rechner (10)

ergänzende Literatur:

- Herrmann: Rechnerarchitektur. Vieweg Verlag
- Märtin: Einführung in die Rechnerarchitektur. Carl Hanser Verlag

Modellierung/Spezifikation

Modellbildung

- Abstraktion (weglassen) von unnötigen Details
- Wahl der geeigneten Darstellung

Realität und Repräsentation:

Realität	Repräsentation
Gegenstände	Zahlen, Wörter
Eigenschaften	Mengen
Zusammenhänge	Relationen

Beispiel: Getränkeautomat

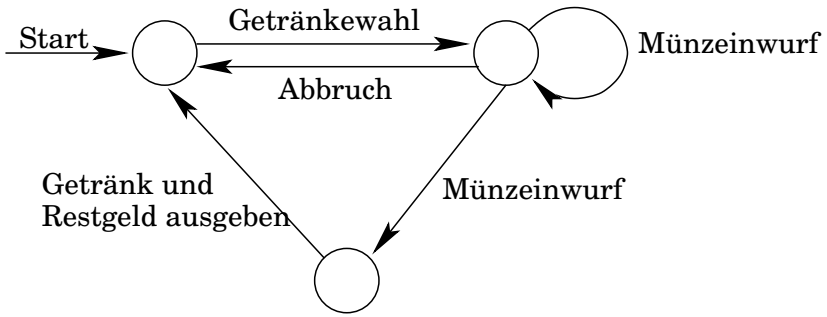
Gegenstände	Cola, Wasser, Bier, 20
Eigenschaften	wählbare Getränke: $\{g_1, \ldots, g_n\}$
Zusammenhänge	Preise: $\{(g_1, p_1), \ldots, (g_n, p_n)\}$

Modellbildung (2)

Es wird abstrahiert von: Farbe und Größe des Automaten, Verfügbarkeit der Getränke usw.

reale Welt $\longrightarrow$ Modellbildung $\longrightarrow$ Repräsentation

Repräsentiert werden müssen Zustände und Operationen.



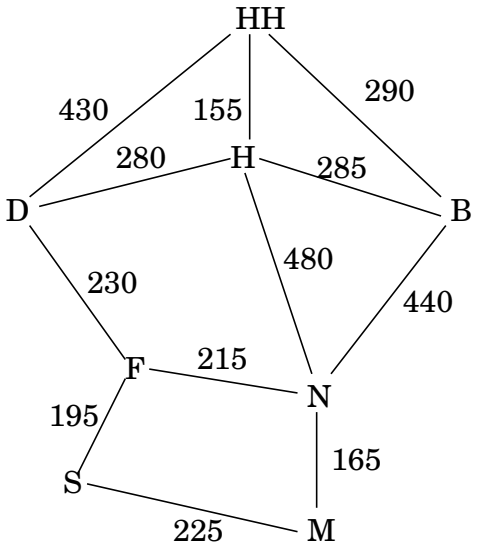
Modellbildung (3)

Beispiel: Wegesuche im Autobahnnetz

Realität	Repräsentation
Entfernung x zwischen A und B	Tripel (A, B, x)
x km Stau zwischen A und B	Tripel (A, B, x)
Baustellen	?
Höchstgeschwindigkeit	?
Wetter	?
Fahrbahnzustand	?
Kürzester Weg	?

Modellbildung (4)

Darstellung eines Verbindungsnetzes:



Gesucht:
die kürzeste Verbindung von Stuttgart nach Berlin.

Modellbildung (5)

Repräsentation im Rechner:

Ort 1	Ort 2	Entfernung
HH	D	430
HH	H	155
HH	B	290
D	HH	430
D	H	280
D	F	230
...	...	...

Ort 1	Ort 2	Entfernung
H	D	280
H	HH	155
H	B	285
H	N	480
F	D	230
F	N	215
...	...	...

Gesucht: Kürzeste Verbindung von Berlin nach Stuttgart.

Spezifikation

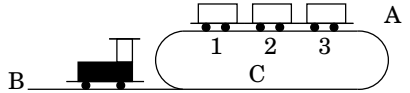
Um einen Algorithmus zu entwickeln, muss das zu lösende Problem genau beschrieben sein.

Anforderungen an eine Spezifikation:

- **vollständig**: Es müssen alle Anforderungen und alle relevanten Rahmenbedingungen angegeben werden.
- **detailliert**: Welche Hilfsmittel bzw. Operationen sind zur Lösung zugelassen?
- **unzweideutig**: Klare Kriterien geben an, wann eine vorgeschlagene Lösung akzeptabel ist.

Spezifikation (2)

Beispiel: Eine Lok soll die Wagen 1, 2, 3 vom Abschnitt A in der Reihenfolge 3, 1, 2 auf Gleisstück C abstellen.



Vollständigkeit:

- Wieviele Wagen kann die Lokomotive auf einmal ziehen?
- Wieviele Wagen passen auf Gleisstück B?

Detailliertheit:

- Was kann die Lokomotive? (fahren, koppeln, ...)

Unzweideutigkeit:

- Darf die Lok am Ende zwischen den Wagen stehen?

Spezifikation (3)

Warum Spezifikation?

- Problemstellung präzisieren
- Entwicklung unterstützen
- Überprüfbarkeit verbessern
- Wiederverwendbarkeit erhöhen

Ziel: Erst denken, dann den Algorithmus entwerfen!

Spezifikation (4)

Bestandteile der funktionalen Spezifikation:

1. Ein Algorithmus berechnet eine Funktion:
   (a) festlegen der gültigen Eingaben (Definitionsbereich)
   (b) festlegen der gültigen Ausgaben (Wertebereich)

2. Funktionaler Zusammenhang zwischen Ein-/Ausgabe:
   (a) Welche Eigenschaften hat die Ausgabe?
   (b) Wie sehen die Beziehungen der Ausgabe zur Eingabe aus?

3. Festlegen der erlaubten Operationen.

Spezifikation: Beispiele

Euklidischer Algorithmus

gegeben: $n, m \in \mathbb{N}$
gesucht: $g \in \mathbb{N}$
funktionaler Zusammenhang: $g = ggT(n, m)$

Jüngste Person

gegeben: $(a_1, \ldots, a_n) \in \mathbb{N}^*, n > 0$
gesucht: $p \in \{1, \ldots, n\}$
funktionaler Zusammenhang:

- $\forall i \in \{1, \ldots, n\}$ gilt: $a_p \leq a_i$

oder alternativ:

- $\forall j \in \{1, \ldots, n\}$ gilt: $a_j \neq a_p \Rightarrow a_j > a_p$.

Verifikation

Ziel: Beweise, dass der Algorithmus korrekt ist!

Wechselspiel zwischen:

- statische Aussagen über den Algorithmus ohne ihn auszuführen → nicht vollständig möglich: zu komplex und umfangreich
- dynamisches Testen des Algorithmus → zeigt nur die Anwesenheit von Fehlern, nicht deren Abwesenheit

Programmverifikation: zeige, dass der Algorithmus die funktionale Spezifikation erfüllt

- der Algorithmus liefert zu jeder Eingabe eine Ausgabe
- die Ausgabe ist die gewünschte Ausgabe

Verifikation (2)

Notation:

- $\{P\}$ Schritt $\{Q\}$
  - P: Vorbedingung
  - Q: Nachbedingung
  - falls vor Ausführung P gilt, dann gilt nachher Q
- $\{P_0\}$ Schritt 1 $\{P_1\}$ Schritt 2 $\{P_2\}$ ... Schritt n $\{P_n\}$
  - falls vor Ausführung des Algorithmus P_0 gilt, dann gilt nachher P_n

Zuweisung: $\{P(t)\} \quad v := t \quad \{P(t) \wedge P(v)\}$
Was vorher für t gilt, gilt nachher für v.

$\{t > 0\} \quad v := t \quad \{t > 0 \ \wedge \ v > 0\}$
$\{x \in \mathbb{N}\} \quad y := x + 5 \quad \{x, y \in \mathbb{N} \ \wedge \ y \geq 5\}$

Verifikation (3)

Wiederholung: $\{P\}$ solange B wiederhole Schritt $\{P \wedge \neg B\}$
P gilt vor und nach jeder Ausführung von Schritt.

$x := y$
$k := 0$
$\{y = k \cdot a + x\}$
solange $x \geq 0$ wiederhole
    $x := x - a$
    $k := k + 1$
$\{y = k \cdot a + x \ \wedge \ x < 0\}$