Grundlagen der Informatik 2010

Prof. Dr. Konrad Froitzheim

Thema

Informatik ist die Wissenschaft von der automatischen Verarbeitung von Informationen

=> Informatik = Information + Automatik

- Information
  - lateinisch: Auskunft, Belehrung, Benachrichtigung
  - umgangssprachlich: Kenntnis über Sachverhalte und Vorgänge
  - inhaltliche Bedeutung eines Sachverhaltes (Semantik)
- Daten
  - Informationen, die zum Zweck der Verarbeitung eine bestimmte, vorher vereinbarte Darstellungsform (Syntax) haben
  - dargestellt durch kontinuierliche Funktionen: "analoge Daten"
  - oder Zeichen: "digitale Daten"
- Automatik (griech.): selbsttätiger Ablauf
- Rechnen
  - addieren, subtrahieren, multiplizieren, dividieren, …
  - konvertieren (umrechnen), bewegen, speichern, …

- Aufgabe der Informatik
  - Erforschung der grundsätzlichen Verfahrensweise der Informationsverarbeitung
  - Entwicklung allgemeiner Methoden
  - Anwendung in den verschiedensten Bereichen
- Informatik als Strukturwissenschaft
  - Struktur und Eigenschaften von Informationen
  - Struktur und Eigenschaften Informationsverarbeitungssystemen
  - heute elektronische Datenverarbeitungsanlagen
  - Methoden: Analyse, formale Beschreibung, Konstruktion
- Informatik ist Ingenieurwissenschaft
  - Computer und Zubehör konstruieren
  - Verfahren und Algorithmen entwerfen
  - Programmieren
  - Pflegen und warten
  - aber: D. Knuth: The Art of Computer Programming

Vision

- Vannevar Bush, 1945, Forderung:

> Memex - a device in which one stores all his books, records, and communications, and which is mechanized so that it can be consulted with exceeding speed and flexibility. It is an enlarged intimate supplement to his memory.

  - Internet + Web: Memex ist implementiert
  - Apple, 1988: Knowledge Navigator - Memex mit verbessertem IF
- Holodeck
  - ca. 1990: Vision in Star Treck
  - Neal Stephenson, 2000: Snowcrash
  - ab 2004: 3D-Welten WoW, SecondLife, …
  - immersive 3D-Interfaces noch teuer -> X-Site
- Alan Turing, 1950, Vorhersage:

> At the and of the century, the use of words and general educated opinion will have changed so much that one will be able to speak of "machines thinking" without expecting to be contradicted.

Gebiete der Informatik

- Theoretische Informatik
  - Berechenbarkeit
  - Aufwand für Berechnung
  - Beweisbarkeit von Programmen
  - Entscheidbarkeit
  - formale Methoden
  - Sprach-Klassen
- Algorithmen und Datenstrukturen
  - Formeln, Ablauf, Sequentialisierung
  - Daten darstellen und speichern
  - Zahldarstellung
- Programmieren
  - vom Algorithmus zum Programm
  - ingenieurmässiges Gestalten (kreativ und systematisch)
  - Software Engineering
  - Programmiersprachen und Compiler
  - Objekte, Patterns und Frameworks

- Betriebssysteme
  - Abstraktion von Geräten und Computer-Teilen
  - Koordination der Programme
  - gerechte Verteilung der Ressourcen (Zeit, Platz, …)
  - Schutz und Sicherheit
- Rechnerarchitektur
  - vom Transistor zum Maschinenbefehl
  - Speicher, Festplatte, Eingabe, Ausgabe, …
  - Parallelrechner
- Technische Kommunikation
  - Signalübertragung und -Vermittlung
  - Protokolle
  - Kompression
  - ISDN, Rechnernetze, Internet, …

- Anwendungen
  - Bürosoftware
  - Datenbanken und Informationssysteme
  - Multimedia
  - Kommunikationsdienste (Telefon, TV, WWW, …)
  - CAD - Computer Aided Design
  - Simulation
  - Spiele
  - Steuerung und Robotik
  - Künstliche Intelligenz
  - Neuroinformatik
- Gebieteeinteilung der Gesellschaft für Informatik (GI)
  - theoretische Informatik
  - praktische Informatik
  - technische Informatik
  - Angewandte Informatik

Formales

Termine:

Vorlesung:	Montag	18:00 - 19:30, Mn-1020
	Freitag	11:00 - 12:30, Mn-1020

Dramatis Personae:

Prof. Dr. Konrad Froitzheim
Professur Betriebssysteme und Kommunikationstechnologien
frz@tu-freiberg.de

Dipl.-Ing. Mathias Buhr

MSc. BNC Frank Gommlich

Vorlesungsunterlagen(kein Skriptum):
http://ara.informatik.tu-freiberg.de/Vorlesung/GdI2010_1.doc
Prüfung: Klausur in der vorlesungsfreien Zeit

Diese Unterlagen zur Vorlesung sind weder ein Skriptum noch sind sie zur Prüfungsvorbereitung ausreichend.

Ohne das gesprochene Wort in der Vorlesung sind sie unvollständig und können sich in Einzelfällen sogar als irreführend erweisen.

Sie dienen dem Vortragenden als Gedächtnisstütze und den Hörern als Gerüst für ihre Vorlesungsnotizen.

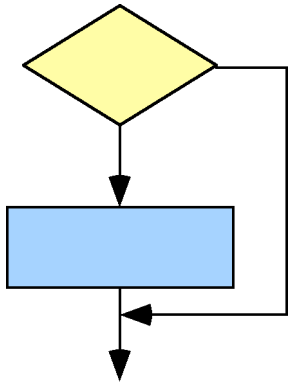
Zur Prüfungsvorbereitung ist die intensive Lektüre der Fachliteratur unumgänglich.

Die Unterlagen werden während des Semesters noch gändert oder ergänzt.

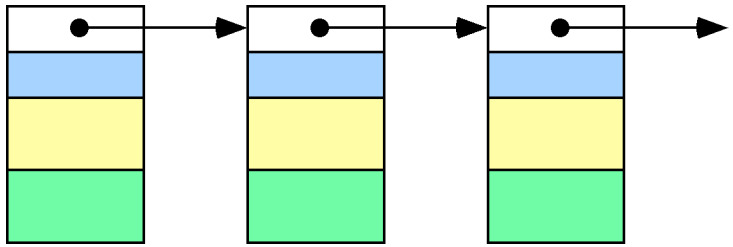
Literatur

Balzert, H.: Lehrbuch Grundlagen der Informatik; Elsevier, 2005.
Eernisse, M.: Build your own AJAX web application; Sitepoint, 2006.
Goll, Bröckl, Dausmann: C als erste Programmiersprache; Teubner, 2003.
Horn, Kerner: Lehr- und Übungsbuch Informatik; 1995.
Kernighan, B., Ritchie, D.: C
Knuth, D.,E.: The Art of Computer Programming; 1973.
Küchlin, W., Weber, A.: Einführung in die Informatik; Springer, 2000
Mark, D., LaMarche, J.: Beginning iPhone 3 Development; 2009
Rembold, U.: Einführung in die Informatik; Hanser, 1987.
Schöning, U.: Theoretische Informatik - kurzgefaßt; Spektrum, 1995.
Sedgewick, R.: Algorithms in C; Addison Wesley, 1998 …
Wirth, N.: Algorithmen und Datenstrukturen; Teubner, 1983 …

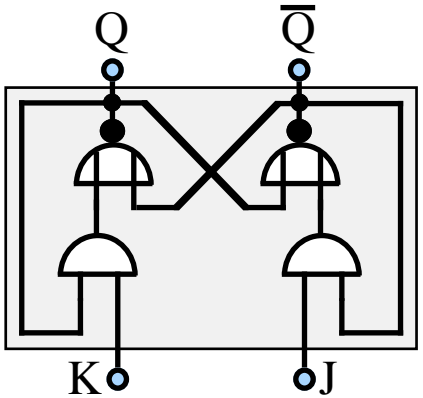
Inhaltsübersicht



3. Algorithmische Komponenten
   - 3.1 Sortieren
   - 3.2 Listen
   - 3.3 Speicherverwaltung
   - 3.4 Rekursion
   - 3.5 Objekte, Patterns und Frameworks



4. Rechnerarchitektur: Vom Transistor zum Programm
   - 4.1 Transistoren, Gates
   - 4.2 Rechnen und Speichern
   - 4.3 Funktionseinheiten: Speicher, Prozessor, Bus, …
   - 4.4 Maschinenbefehle
   - 4.5 Compiler



5. Betriebssystem: Abstrahieren und Koordinieren
   - 5.1 Verteilung der Ressourcen
   - 5.2 Struktureller Aufbau am Beispiel iOS
   - 5.3 Toolbox/Frameworks am Beispiel iOS

6. Medien
    6.1 Medien und Wahrnehmung
    6.2 Computergrafik
    6.3 Standbilder
    6.4 Video
    6.5 Audio
7. Eine eigene App
8. Theoretische Informatik
    8.1 Automaten und formale Sprachen
    8.2 Berechenbarkeit
    8.3 Komplexitätstheorie
8. Anwendungsprogramme
    8.1 Datenbanken
    8.2 Bürosoftware
    8.3 Medienbearbeitung

1 Information und Daten

1.1 Informationsbegriff

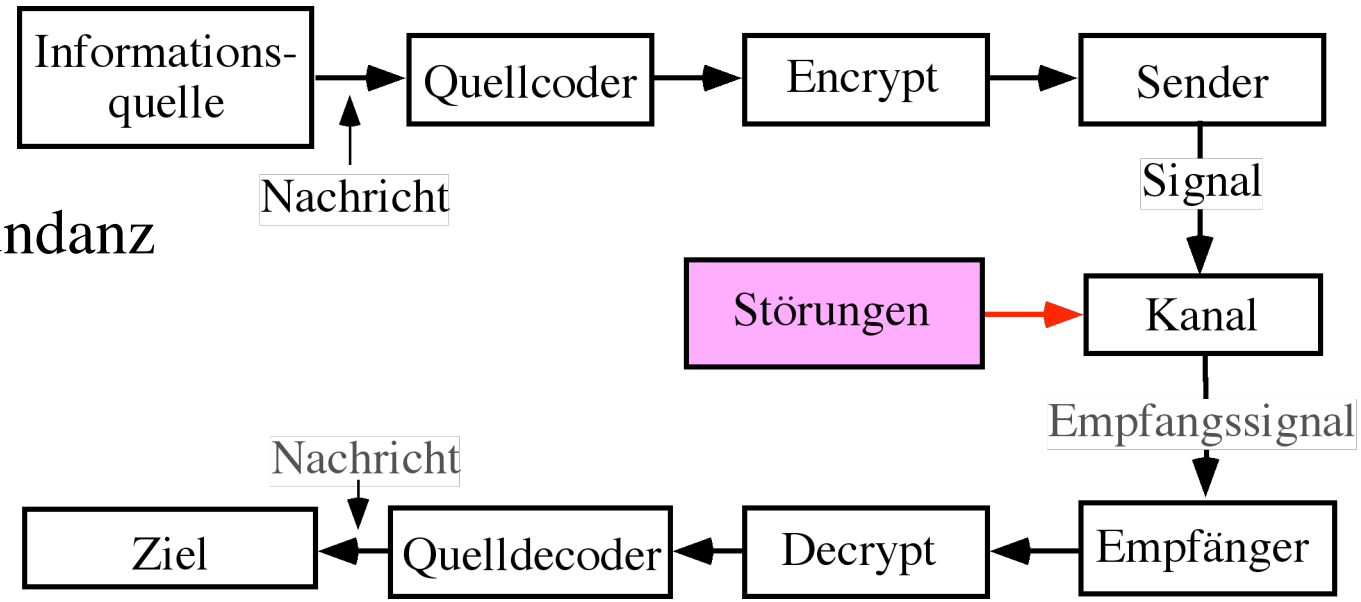
- Shannon, 1948: Definition als mathematische Größe
- Nachrichtenquelle
  - Nachrichten mit Wahrscheinlichkeit $p(s_i)$
  - Wahrscheinlichkeit bestimmt Informationsgehalt
- Informationsgehalt eines Symbols: (Selbstinformation) $I_i = -ld\ p(s_i)$ bit
  - Entropie einer Nachrichtenquelle $H = \sum_i p(s_i) \bullet ld \frac{1}{p(s_i)}$
- Beispiel Münzwurf
  - Ereignisse Kopf und Zahl
  - p(Kopf) = 1/2, p(Zahl) = 1/2

  => H(Kopf) = -ld 1/2 bit = ld 2 = 1 bit
- Beispiel Würfel
  - Ereignisse 1, 2, 3, 4, 5, 6
  - p(i) = 1/6

  => H("3") = -ld 1/6 bit = 2,58… bit

- Wahrscheinlichkeit einer Symbolfolge $s_1 s_2 \ldots s_n$
  - Symbole haben Wahrscheinlichkeit $p(s_i) = p_i$
  - $p = p(s_1) * p(s_2) * \ldots p(s_n)$
  - p("the") = 0,076 * 0,042 * 0,102 = 3,25584 * 10^{-4}
- Wahrscheinlichkeit $p(s_i)$
  - allein ≠ im Kontext
  - $p(s_i = \text{'h'}) = 0{,}042$ => H('the') = 11,6 bits
  - $p(s_i = \text{'h'} \mid s_{i-1} = \text{'t'}) = 0{,}307$ => H('the') = 6,5 bits
- Informationsrate in
  - englische Buchstaben: 4,76 bit/Buchstabe
  - englischem Text: 4,14 bit/Buchstabe
  - englischem Text: 9,7 bit/Wort, ~ 2 bit/Buchstabe
- Modell entscheidend

- Satz von der rauschfreien Quellkodierung
  - $H(N) = x$ bit $\Rightarrow \exists$ Kode K mit $Länge_K(N) = x+\varepsilon$ Bits
  - Mittlere Kodelänge kann Entropie annähern
- Kode = Nachricht + Redundanz
- Übertragung
  - zwischen Computern
  - im Computer
  - Speicherung



- Kommunikation [Shannon, Weaver]:
  - Lebewesen oder Maschine beeinflußt anderes Lebewesen oder Maschine
  - technisches Problem
  - semantisches Problem (Symbole und ihre Bedeutung)
  - Effektivitäts-Problem (Einfluß der Bedeutung)

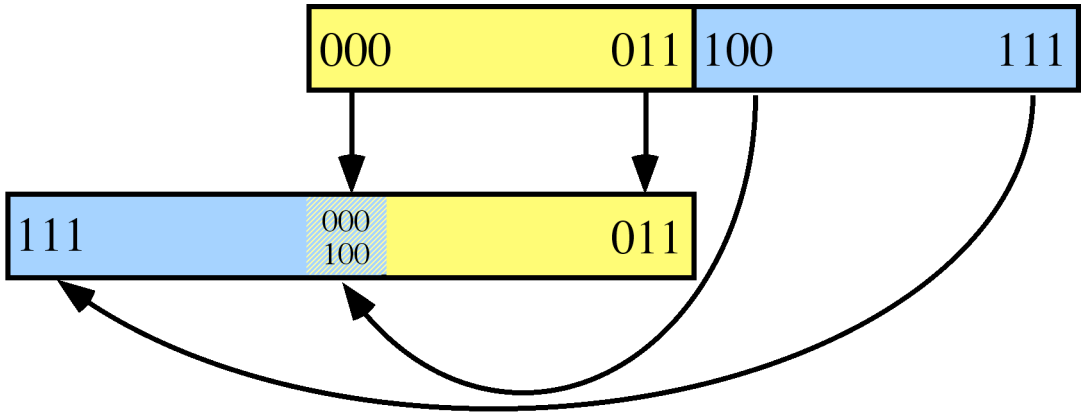
1.2 Zahldarstellung

1.2.1 Ganze Zahlen

- Polyadisches Zahlsystem
  - $z = \sum_{i=0}^{n-1} a_i B^i$
  - $0 \le a_i < B$
  - Basis 10: $1492 = 2*10^0 + 9*10 + 4*100 + 1*1000$
  - Basis 2: $1492 = 1*1024+0*512+1*256+1*128+1*64$
    $+0*32+1*16+0*8+1*4+0*2+0*1 = 10111010100$
  - Basis 16: $1492 = \$5D4 = 5*256 + 13 * 16 + 4 * 1$
- Zahlenbereich
  - 10 Bit => 0..1023 = 1 'Kilo' -1
  - 20 Bit => 0..1024*1024-1 = 1 'Mega' -1
  - 32 Bit => 0..4 294 967 295 (2^{32}-1) = 4 'Giga' - 1
- negative Zahlen?

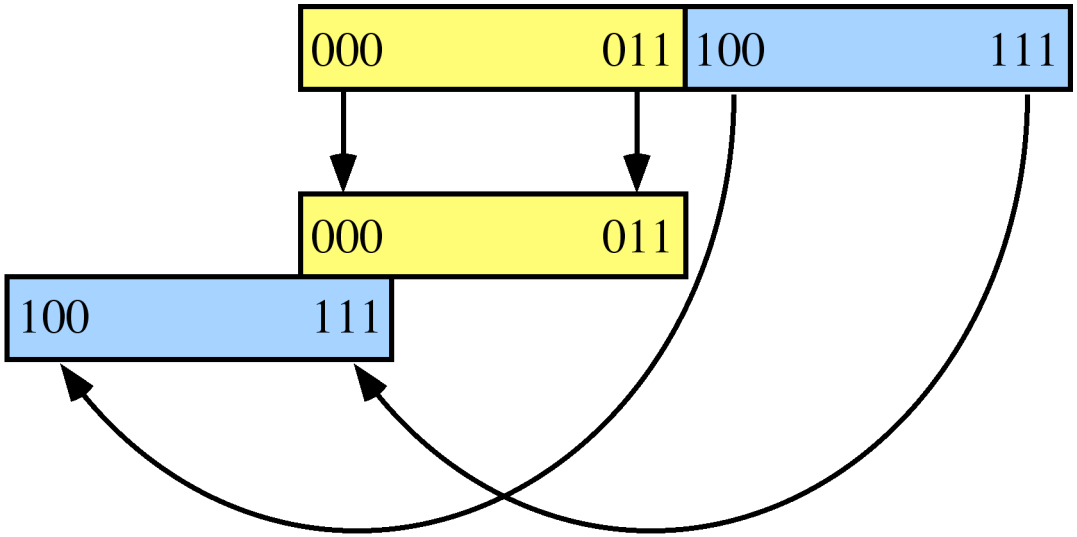
- Vorzeichen-Betrag (sign-magnitude)
  - 1 Bit Vorzeichen
  - höchstes Bit

  - 8 Bit (256 Werte): -127..127



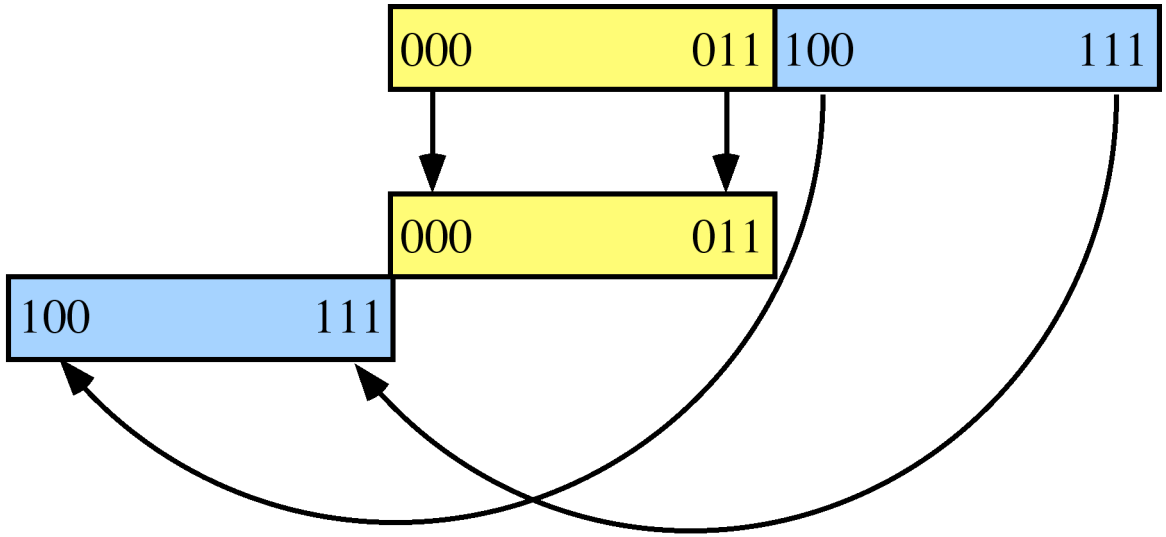
  - komplexe Rechenregeln

- Stellenkomplement
  - jede Stelle 'negieren': 0->1, 1->0
  - 00111111 = 63
  - 11000000 = -63



  - negative Null: 00000000 und 11111111
  - -127..0,0..127
  - besondere Rechenregeln

- Basiskomplement
  - jede Stelle negieren, 1 addieren
  - 00111111 = 63
  - 11000000+1= 11000001= -63



  - nur eine Null
  - Umrechnung macht mehr Arbeit
  - Rechenregeln einfach

1.2.2 Rationale und Reelle Zahlen

- Rationale Zahlen
  - Brüche: 1/2, 1/3, …
  - Dezimalschreibweise ggg,ddd: 0,5; 12,625
  - evtl unendlich viele Stellen: 16/3
  - ggggg,ddddd… : 1,33333333333333333333...
  - ggg,ddd = ggg + 0,ddd
  - ganzzahliger Teil + Bruchteil

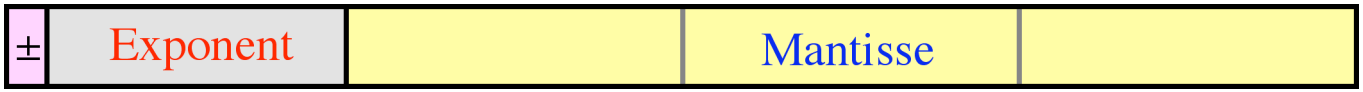
±	64	32	16	8	4	2	1

$\frac{1}{2}$	$\frac{1}{4}$	$\frac{1}{8}$	$\frac{1}{16}$	$\frac{1}{32}$	$\frac{1}{64}$	$\frac{1}{128}$	$\frac{1}{256}$

- Näherungsdarstellung von reellen Zahlen
  - $\pi \approx 3{,}1415926$
  - $\sqrt{2} \approx 1{,}414213562$
  - wann sind diese Fehler nicht katastrophal?

  => numerische Mathematik

- Normalisieren
  - ggg,ddd = 0,gggddd * 10^{3}
  - 1,34567 = 0,134567 * 10^{1}
  - 0,012345 = 0,12345 * 10^{-1}
  - 0,0000001 = 0,1 * 10^{-6}
- Floating-Point Zahlen
  - 0 < Mantisse < 1
  - 10^{exp} "Exponent"
  - Vorzeichen



- Trick
  - normalisieren => 0,10111010101010 *2^{exp}
  - erstes Bit immer = 1 => weglassen
- typische Formate
  - ANSI/IEEE 754-1985
  - single: 32 bit - 23 bit Mantisse, 8 bit Exponent
  - double: 64 bit - 52 bit Mantisse, 11 bit Exponent
  - extended: 80 bit

1.3 Rechnen

- Addieren im Zehnersystem
  - stellenweise addieren

```
   1513
+  2112
------
   3625
```

  - Übertrag

```
   1523
+  2192
------
   3715
```

  - Rechenregeln für Übertrag 7+8 = 5 + Übertrag 1
- Binärer Fall
  - stellenweise addieren

```
   01001010
+  00101111
-----------
   01111001
```

  - Rechenregeln einfach: 0+0=0, 0+1=1, 1+0=1, 1+1=0 Übertrag 1

- Beschränkte Stellenzahl
  - größte darstellbare Zahl
  - Ergebnis grösser: Überlauf

```
    11001010
+   10101111
    ---------
    011110001
```

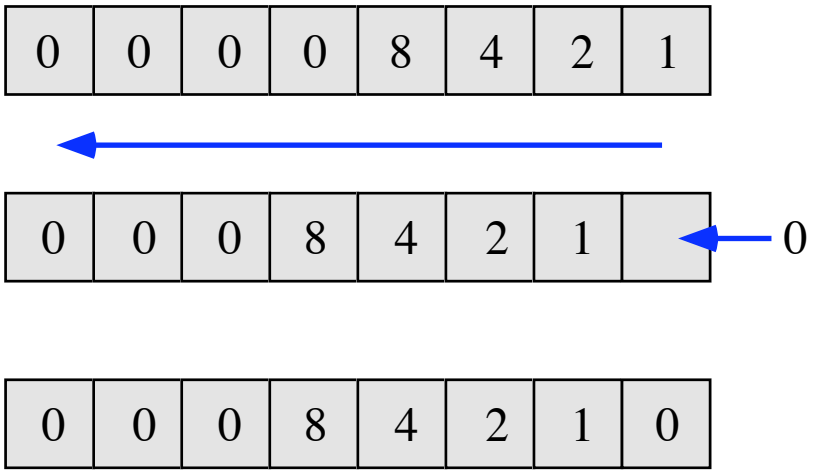
  - eventuell negative Zahl

```
    01001010
+   01101111
    ---------
    101110001
```

- Subtrahieren
  - Komplement addieren: a-b = a+(-b)
  - besondere Regeln zur Ergebnisinterpretation

- Multiplikation
- Primitives Verfahren
  - Multiplikand * Multiplikator
  1. erg auf Null setzen
  2. erg = erg + Multiplikand
  3. Multiplikator um 1 herunterzählen
  4. falls Multiplikator > 0: weiter mit 2
- Sonderfall
  - Verschieben um eine Stelle, Null nachziehen

  => Multiplikation mit Stellenwert



- Multiplizieren
  - Multiplikand * Multiplikator
    1. i auf Eins setzen
    2. Addieren von (Multiplikand * Stelle i des Multiplikators)
    3. Verschieben des Multiplikanden (mit Stellenwert multiplizieren)
    4. i hochzählen
    5. weiter mit 2, falls i ≤ Stellenzahl Multiplikator
  - im Zehnersystem:

$$
\begin{array}{r}
1214 * 211 \\
\hline
1214 \\
12140 \\
242800 \\
\hline
256154
\end{array}
$$

- Trick:
  - Multiplikator in jedem Schritt eine Stelle nach rechts schieben
  - letzte Stelle in Schritt 2 verwenden

- binär Multiplizieren
  - ShiftLeft = Multiplikation mit 2
  - Stellen nur 0 oder 1 => Multiplikand addieren oder nicht
- Verfahren
  1. Ergebnis = 0; i = 0; a = Multiplikand; b = Multiplikator
  2. falls letzte Stelle von b = 1: Ergebnis = Ergebnis + a
  3. ShiftLeft(a)
  4. ShiftRight(b)
  5. Falls b>0 : weiter mit 2
- Beispiel: 12 * 5

Iteration	a	b	erg
0	0000 1100	0000 0101	0000 0000
			+ 0000 1100
1	0001 1000	0000 0010	0000 1100
			nix tun
2	0011 0000	0000 0001	0000 1100
			+ 0011 0000
3	0110 0000	0000 0000	0011 1100

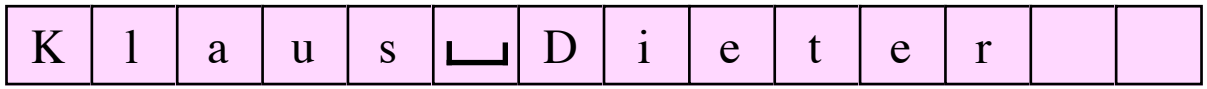
1.4 Datentypen

- Buchstaben
  - 'A', 'B', …, 'Z', 'a', 'b', …, 'z', '.', ',', ';', …, '0', …, '9'
  - 65, 66, …
  - ASCII, EBCDIC
  - Sonderzeichen: .,!"§$%&/()=?@”#^\˜·¯¨*#;:_-<>
  - unsichtbare Zeichen: Space/Blank, CR

```
char <Name>;
```

- Zeichenketten (strings)
  - zusammengesetzt aus Buchstaben
  - 'Auto', 'Bergakademie', 'Klaus Dieter'

```
char <Name> [n];
```



- Wie lang ist ein string?
  - Länge(string) = Anzahl Buchstaben
  - Länge('Auto') = 4
  - Länge('Klaus Dieter') = 12
  - Längenfeld
  - oder besonderes Zeichen am Ende

- Boolsche Typen
  - 2 Werte: true, false
- Aufzählungen
  - frei gewählte Bezeichner
  - (rot, gruen, blau)
  - (Mercedes, BMW, VW, Ford, Opel, Audi, Porsche)
  - werden auf Zahlen abgebildet
  - `enum {red,green,blue} color`
- Zahlen (typische Länge …)
  - `short, unsigned short (16 bit)`
  - `int, unsigned int (16, meist 32 bit)`
  - `long, unsigned long (32 oder 64 bit)`
  - `float (32 bit)`
  - `double (64 bit: 2.22 e-308 bis 1.79 e+308)`

- Zusammengestzte Datentypen
  - aus mehreren elementaren Datentypen zusammengesetzt
  - gefühlter Sonderfall Zeichenkette siehe oben
- Vektoren und Matrizen

7	12	73	0	0	0	123	456	13	13	12	9	12	9

  - gleiche Datentypen

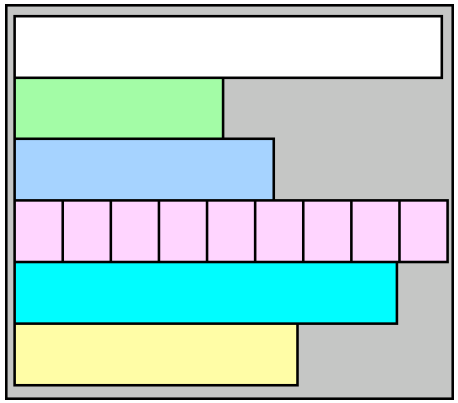
$$\begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix} \qquad \begin{pmatrix} 1 & 0 & 7 & 8 \\ 0 & 1 & 9 & 2 \\ 12 & 13 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \qquad \begin{pmatrix} \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 1 \end{pmatrix} & \begin{pmatrix} 1 & 2 & 4 \\ 4 & 5 & 6 \\ 7 & 8 & 4 \end{pmatrix} \\ \begin{pmatrix} 7 & 2 & 5 \\ 4 & 5 & 6 \\ 7 & 8 & 4 \end{pmatrix} & \begin{pmatrix} 4 \\ 6 \\ 8 \end{pmatrix} \end{pmatrix}$$

```
float vielstufig [n][m]…[k]
```

- Datenstrukturen (Records, zusammengesetzte Daten, …)
  - unterschiedliche Typen

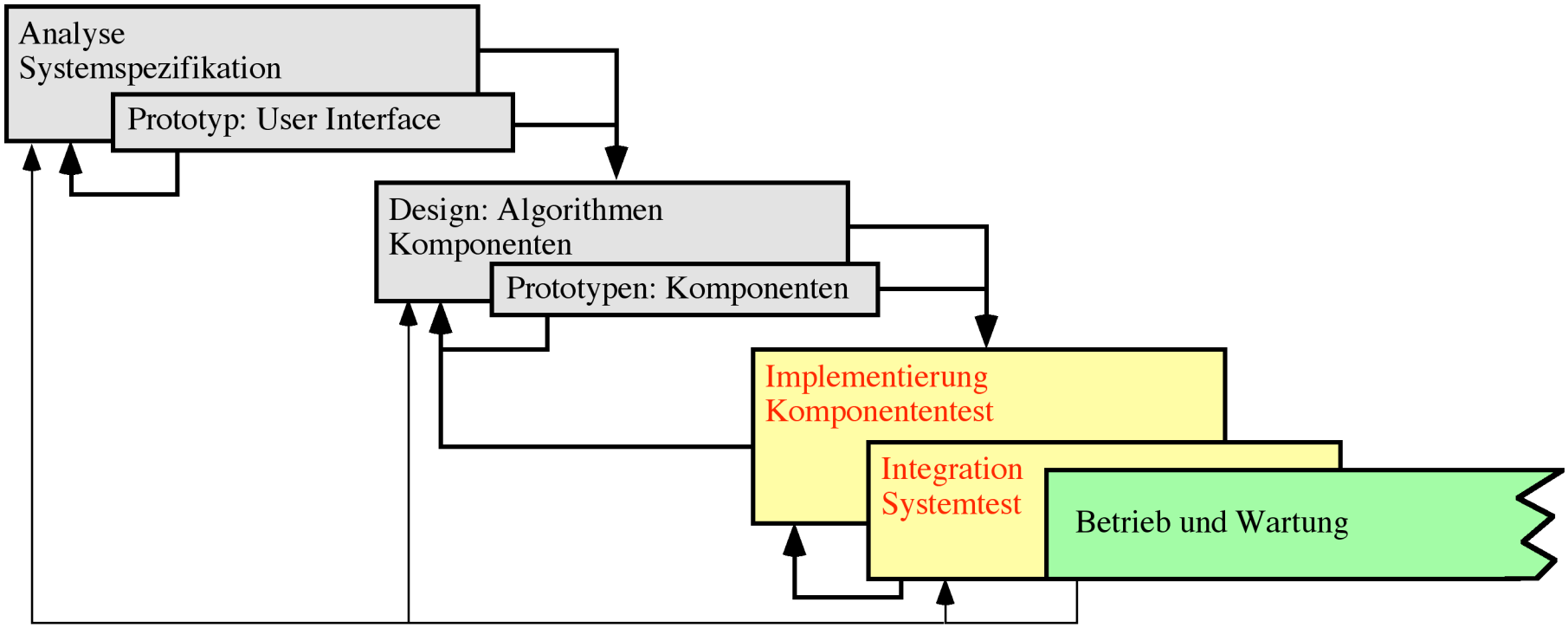
```
struct Wagenbeschr
{  char     name[32];
   enum     {Schlaf,Abteil, …} Wagentyp ;
   int      Baujahr;
};
```

  - beschreibt Gegenstände, Personen, …



2. Vom Problem zum Programm

- The Art of Computer Programming [Knuth, 1968]
- Programmieren ist Ingenieurtätigkeit
  - systematisch, planbar
  - reproduzierbar
  - dokumentiert
- Theoretischer Ablauf



- Computer Aided Software Engineering (CASE)
  - Computerprogramm zum Programmieren
  - grafische Hilfsmittel
  - formale Spezifikation
  - Verifikation der Quer-Einflüsse
  - Versionsverwaltung
- Rollen
  - Projektleiter
  - Analytiker
  - Programmierer
  - Tester
- Phasenmodell
  - globale Anforderungen, Struktur der Arbeit
  - Untersuchung eines Geschäftsgebietes, Anforderungen
  - Systementwurf aus Benutzersicht
  - Technische Design
  - Codierung, Datenbank-Generierung
  - Systemeinführung, Test
  - Verwendung (Produktion)

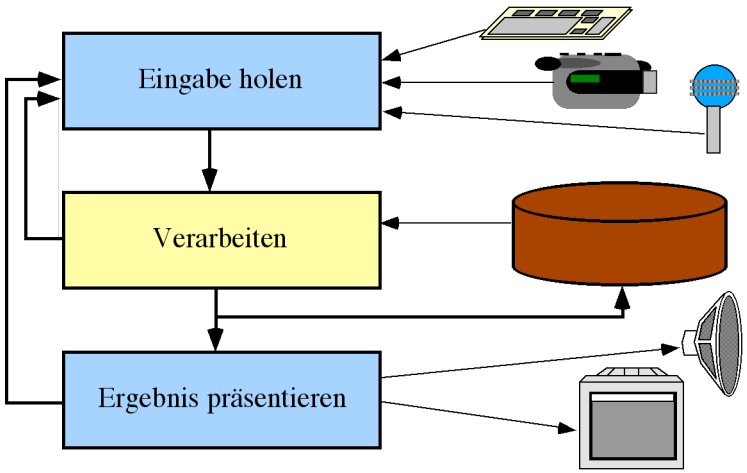
- Agile software development
  - kleine Gruppen
  - Selbstanordnung
  - ständige Kommunikation
  - intensive Reviews
  - Kunden integriert in Zyklus
- Extreme Programming

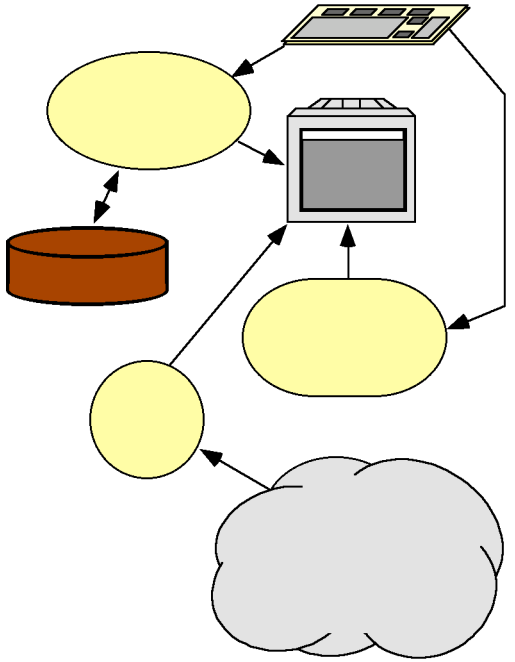
- Imperatives Programmieren
  - Programm entsprechend Datenfluss
  - Funktionen zur Daten- und Zustandänderung
  - Datenstrukturen
  - Pascal, C, Modula, Cobol, Fortran, …
- Objektorientiertes Programmieren
  - Datenstrukturen mit Transformationsfunktionen
  - Zustände in den Datenstrukturen
  - ereignisorientiertes Programmieren
  - Smalltalk, Java, C++, Oberon, …
- Funktionales Programmieren
  - Folgen mathematisch/logischer Funktionen
  - Lambda-Kalkül
  - ohne Zustände und änderbare Daten
  - APL, ML, **Haskell**, Lisp, Prolog, …

```
fac = 1; i = 1;
do
    {fac = fac * i;
     i = i+1;}
while (i <= n);
```

```
fac :: Integer -> Integer
fac 0 = 1
fac n | n > 0 = n * fac (n-1)
```

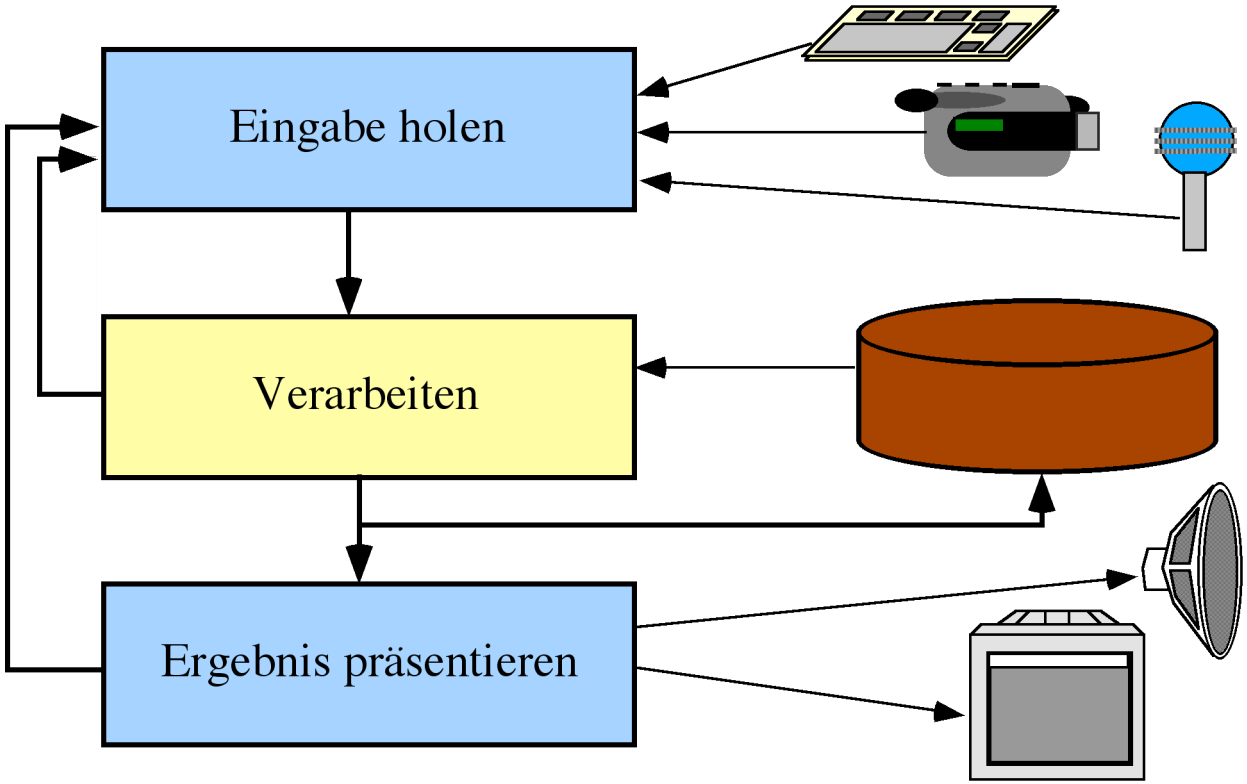
- Aktives Programmieren
  - Ablauf kontrollieren
  - Beginn,
    {Eingabe, Verarbeitung, Ausgabe},
    Ende
  - Programmierer plant Programmablauf
  - Programmiersprachen siehe oben
- Reaktives Programmieren
  - Komponenten bereitstellen
  - gekapseltes Verhalten
  - ereignisgesteuert
  - Anreiz -> Transformation -> Antwort
  - AJAX
  - php, Ruby, …





2.1 Idee und Analyse

- Aufgaben
  - Berechnen
  - Bearbeiten
  - Organisieren
  - Kommunizieren
  - Messen und Steuern
- Programmstruktur



- Problem definieren
  - intuitve Beschreibung
  - Eingabe detailliert aufschreiben
  - Resultate definieren
  - Ausgabe genau aufschreiben

Problem	Verfahren	typische Operationen
Pullover anfertigen	Strickmuster	rechte Masche, linke Masche
Kuchen backen	Rezept	Mehl wiegen, rühren
Schrank aufstellen	Bauanleitung	schrauben, stecken, fluchen

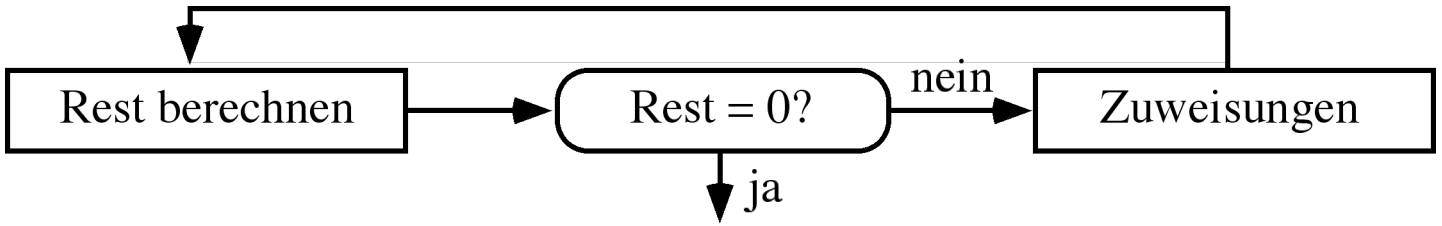
- Wissen im Gebiet sammeln
  - Erfahrung der Mitarbeiter bzw. Kunden nutzen
  - klassiche Bearbeitung genau studieren
  - ähnliche Programme untersuchen
  - Gesetze bzw. Normen beachten
- Verfahren (er)finden
  - Formeln und Regeln
  - Menge von Operationen

- Berechenbarkeit prüfen
  - Eingabewerte erfassbar?
  - Ergebnis ausdrückbar?
  - Laufzeit (Komplexität) des Verfahrens
  - Ressourcen (Speicherplatz)
- Datenstrukturen bilden
  - Abbildung der Eingabe
  - Transformation in Ausgaben
  - Vollständigkeit
  - Speicherplatz
  - optimierter Zugriff (schnell und/oder einfach)
- Werkzeuge auswählen
  - Computertyp- und Ausstattung, Betriebssystem
  - Programmierwerkzeuge (Sprache, Prototyping-System, …)
  - Basissoftware (Datenbank, …)
- Kosten ermitteln
  - Arbeitszeit
  - Geräte und Material

2.2 Sequentialisierung

- Algorithmus
  - Abu Ja'far Mohammed ibn Mûsâ al-Khowârizm
  - Abfolge von Schritten
  - Rechenvorschrift
  - bis ca. 1950: nur im Zusammenhang mit Euklidscher Algorithmus
- Euklidscher Algorithmus
  - größten gemeinsamen Teiler ggt von (m,n) finden

```
1. setze rest = (m DIV n)
2. rest = 0? setze ggt = n; fertig
3. setze m = n und n = rest; weiter mit 1.
```



- Schrittweise Verfeinerung (step-wise refinement)
  - Zerlegung in Teilprobleme
  - Verfahren finden oder entwickeln

- Beispiel: Kaffee machen (Pseudocode)
- Grobverfahren

```
1. Wasser kochen
2. Nescafe in die Tasse
3. Wasser in die Tasse
```

- Einzelschritte verfeinern

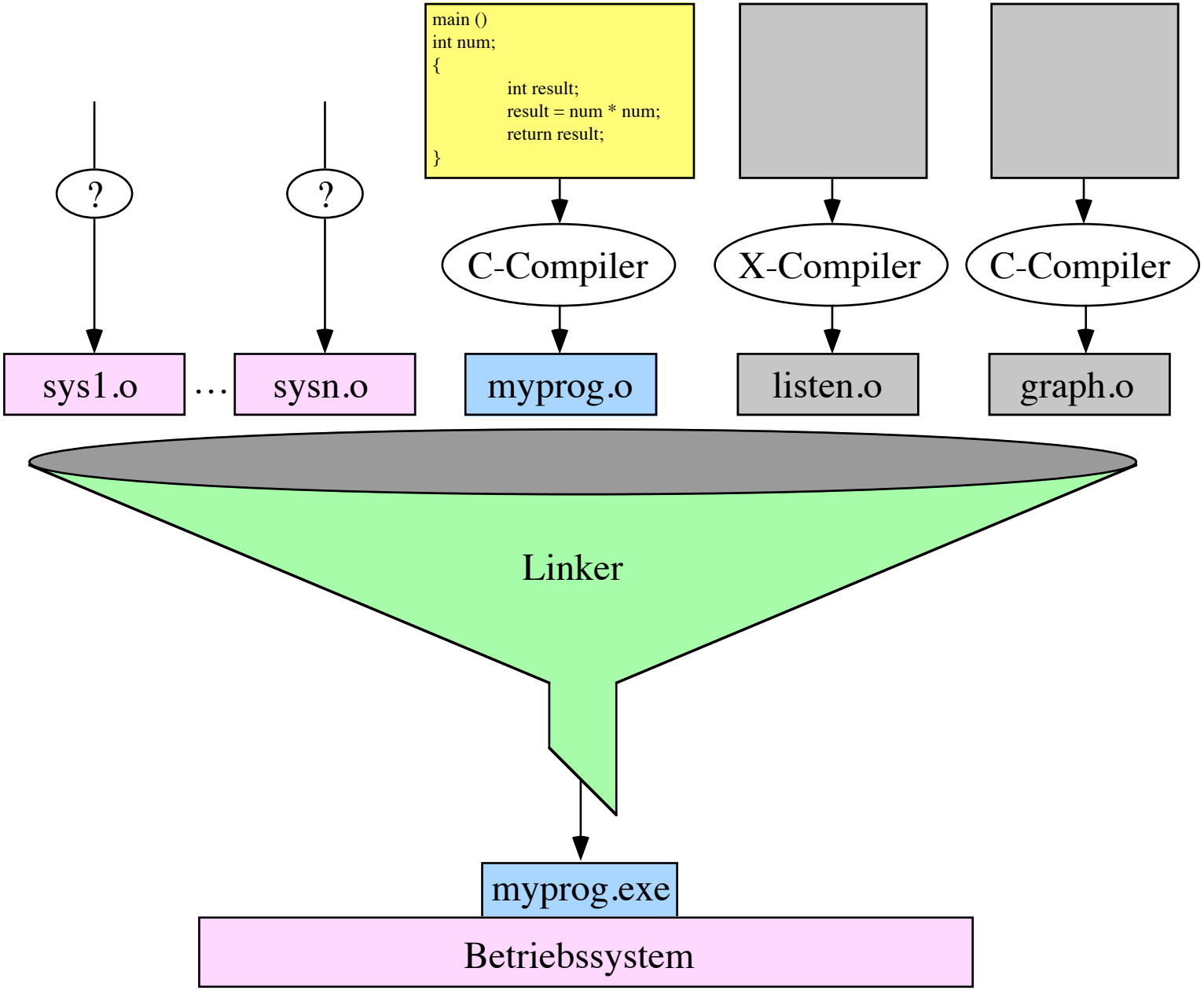
```
1.1 Kessel mit Wasser füllen
1.2 Auf die Herdplatte stellen
1.3 Herdplatte anstellen
1.4 Warten bis Wasser kocht
1.5 Herdplatte abstellen

2.1 Glas öffnen
2.2 Menge Kaffeepulver auf den Löffel laden
2.3 Löffel in die Tasse leeren

3.1 Kessel von der Herdplatte nehmen
3.2 Tasse mit Wasser füllen
```

- Weiter verfeinern

- Methoden zur Strukturbeschreibung
  - Algorithmus beschreiben
  - Verfeinerungsstufen
- Grafische Darstellungsformen
  - Flußdiagramme (DIN 66 001)
  - Struktogramm (Nassi-Shneiderman-Diagramm, DIN 66 261)
  - Datenflußplan
- Sprachliche Beschreibung
  - Pseudocode
  - spezielle Entwurfssprachen
  - Spezifikationssprache
  - Elemente einer Programmiersprache
- Wesentliche Verarbeitungselemente
  - Sequenz
  - Selektion, Iteration
  - Gruppierung, Parallelität
  - Ein- und Ausgabe

2.3 Programmieren

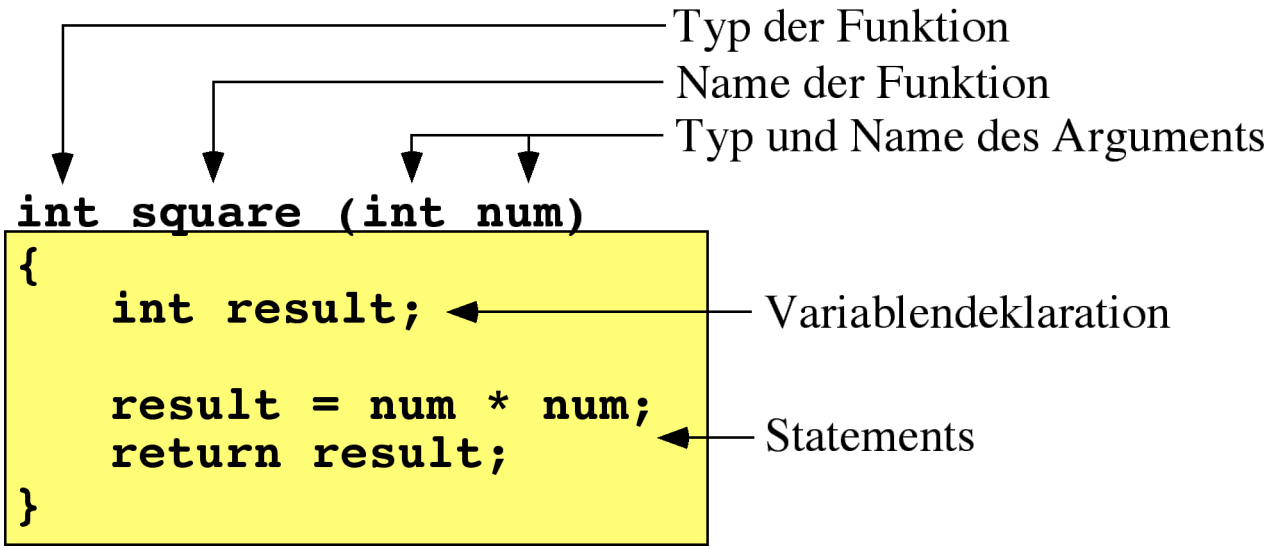
- Viele Programmiersprachen
  - Allzwecksprachen: C, Pascal, Modula, Oberon, Ada
  - Spezialisiert: COBOL, FORTRAN
  - BASIC
  - objektorientiert: SmallTalk, Java, C++,
- Arbeitszyklus
  1. `Editieren`
  2. `Übersetzen (Compiler)`
  3. `Zusammensetzen mit Standardteilen (Linker)`
  4. `Ausführen und Fehlersuchen`
  5. `if Fehler then 1.`
- Fehlersuche
  - wesentlicher Bestandteil des Programmierens
  - Debugger
  - schrittweise ausführen
  - Werte und Zwischenergebnisse amzeigen


main ()
int num;
{
int result;
result = num * num;
return result;
}
?
?
C-Compiler
X-Compiler
C-Compiler
sys1.o
…
sysn.o
myprog.o
listen.o
graph.o
Linker
myprog.exe
Betriebssystem

2.3.1 Grundbegriffe von C

- C
  - von Dennis Ritchie und Ken Thompson
  - weit verbreitet, sehr portabel
  - eng verbunden mit dem Erfolg von UNIX
  - ANSI-Standard X3.159-1989, C89/C90
  - ISO-Standard ISO 9899:1999 (C99)
  - hat vielfältige Ausdrucksmöglichkeiten
  - effizientes Programmieren möglich
  - verleitet zum Tricksen
- Beispiel: eine einfache Funktion

```
int square (int num)
{
   int result;
   result = num * num;
   return result;
}
```



- Funktionskopf (Deklaration)
  - beschreibt die Schnittstelle zu den Benutzern
  - Parameteranzahl und -typ (Argumente)
  - Ergebnistyp
- Funktionsrumpf (Definition)
  - Variablen-Deklaration (Speicherplatz-Bestellung)
  - Verarbeitung
  - benutzt eventuell andere Funktionen

- 3 Funktionen die man immer braucht
- Einlesen von Eingabewerten: `scanf()`
  - beliebig viele Parameter
  - int, float, char, string…
  - von stdio - Standard-Eingabe
  - meist Tastatur

```
scanf("%d %d",&ersteZahl, &zweiteZahl);
```

- Eingabesteuerung
  - Formatstring mit Beschreibung der Eingabe
  - %d, %i: integer
  - %e, %f, %g: float
  - %c: char
  - %s: string
- Besondere Zeichen
  - space trennt und wird übersprungen
  - Zeilenende beendet Eingabe

```
123__456__<CR>  => ersteZahl=123; zweiteZahl=456;
```

- Eingabeanleitung muß mit printf() erzeugt werden

- Ausgabe: `printf()`

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

  - beliebig viele Parameter
  - Formatsteuerung wie scanf()
  - auf "stdout" - Standard Ausgabe
  - z.B. Bildschirm
  - Mischt Formatstring und Variablen

  ```
  printf("hier sind 3 Zahlen: %f %d %f",fl,gz,my);
  ```

- Kombinierte Ein/Ausgabe

  => Benutzungsoberfläche

  ```
  printf("Buchstaben eingeben:"); /* scanf flushed */
  scanf("%c",&zeichen); /* stdin */
  printf("Der Zeichenwert ist: %d\n", zeichen)
  ```

```
Buchstaben eingeben:A
Der Zeichenwert ist: 65

_
```

- Spezial-Funktion `main()`
  - dahinter steckt das selbstgeschriebene Programm
  - gesamte Software ist eine Menge von Funktionen
  - Anwendungsprogramm ist auch eine Funktion

```
#include <stdio.h> /*das Noetigste importieren*/
int main(void)
{
printf("kuckuck\n"); /* Verarbeitung */
return 0;/* mit Rueckgabewert "kein Fehler" */
}
```

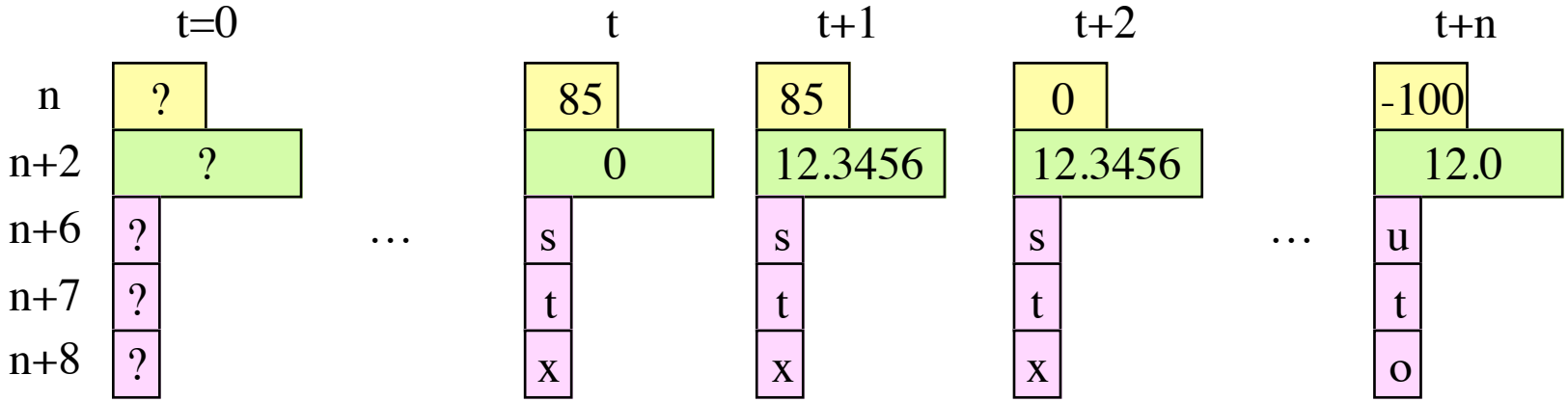
- Programm eintippen mit irgendeinem Editor
  - Notepad, TextEdit, emacs, vi, xedit, …
- Übersetzen und ausführen

```
>cc -Wall -o myprogram myprogram.c
>./myprogram
```

  - cc ist der compiler, linkt auch

2.3.1.1 Variablen-Deklaration: `int result;`

- Identifier
  - Namen, die der Programmierer erfindet
  - Buchstaben, Ziffern, "_", keine Ziffer am Anfang
  - case-sensitive
- Manche Zeichenketten bereits besetzt
  - Schlüsselworte: `if`, `switch`, `while`, …
  - vordefinierte Typen: `int`, `float`, …
- Variablen
  - Container zum Aufbewahren von Werten
  - werden vom Compiler als Speicherplatz angelegt



  - nach dem Programmende verloren

- Mit Identifiern bezeichnet
  - Verwendung in Formeln, …
  - Speicherplatzadresse wird manchmal benötigt
  - `&<ident>` liefert Adresse
- Aussagekräftige Namen
  - dokumentieren Programm
  - lassen Typ erkennen
  - ungarische Notation [Symoni, 1999]
- Deklaration
  - Typ <Name>, …, <Name>;

```
int     i,j;
float   pi, psi, karl_otto;
char    zeichen;
char    name [32], name2 [16], _2tername [31];
```

- Literale
  - Wert fest (Konstante?)
  - für Vergleiche, Initialisierung
  - Zahlen, Buchstaben, String

```
index = index + 1;
if (zeichen > 'Z') …;
zeichen = zeichen + 32; /* macht Kleinbuchst. */
pi = 3.1415926;
mpi = -3.1415926;
```

- Deklaration mit Vorbesetzung
  - Typ <Name> = <Ausdruck>;

```
int     i = 12;
double  pi = 3.1415926;
double  pi1 = 4*atan(1);
double  pi2 = -mpi;
char    space = ' ';
```

- Typkonvertierung
  - zur Anpassung von Parametern
  - Verwendung in Ausdrücken; Bsp: `erg = i * pi;`
  - eventuell mit echter 'Umrechnung'
  - manche Konvertierung nur 'formal'
- Explizite Konvertierung (typecast)
  - `(<typname>) <ausdruck>`
  - `int n = 3; …; erg = sin((double) n);`
  - guter Stil
- Implizite Konvertierung
  - vom Compiler eingebaut
  - typisch in Ausdrücken und Funktionsaufrufen
  - Rückgabewerte
  - Typhierarchie
- Vorsicht mit der impliziten Konvertierung

```
i = 1.5;   /* i enthaelt nun 1 */
j = -5.8;  /* j enthaelt nun -5 */
f = 2;     /* f enthaelt nun 2.0*/
```

- Gültigkeit der Variablen
  - Ort der Vereinbarung
  - im 'Programm' => globale Variable
  - im Block => lokale Variable
  - in der Funktion => lokale Variable
  - entscheidet über Benutzbarkeit
- Lebenszeit von Variablen
  - lokal: in der Funktion - nur während des Funktionsaufrufes (Block)
  - global: während der gesamten Programmausführung
  - static macht auch lokale Variable permanent
  - siehe auch Kapitel Funktionen

2.3.1.2 Operatoren, Ausdrücke und Anweisungen

- Formeln

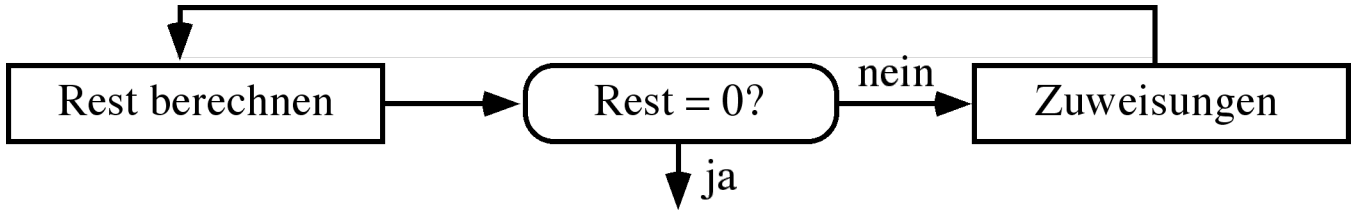
```
dreiecksflaeche = grundlinie * hoehe / 2;
kreisflaeche = radius*radius*3.1415926;
umfang = 2*radius*pi;
```

- Unäre (einstellige) Operatoren

Vorzeichen	+,-
Inkrement und Dekrement	++, --
Adresse	&
Negation	!

- Binäre (zweistellige) Operatoren

Multiplikation	*, /, % (Rest der int-Division)
Addition	+, -
Relation	<, <=, >, >=
Gleichheit	==, !=
logisches UND	&&
logisches ODER	||
Zuweisung	=, +=, -=, *=, …



- %

```
do
{  rest = m % n;
   if (rest != 0)
   { m=n; n=rest;
   }
} while (rest>0);
```

- Vergleiche <, <=, >, >=
  - Zahlen 0 oder 1 als Ergebnis
  - 0 ist false, alles-ungleich-0 ist true

```
rest>0; m!=n;
```

- Gleich oder ungleich: `==, !=`
  - geht oft nicht bei Gleitpunktzahlen
  - liefert 0 oder 1

```
-1 < 0, 0 == 0, 1 != -1     /* true */
0>1, 1>10                   /* 0 */
```

- Logische Operatoren: `&&, ||`

```
((a>b)&&(f>g))  /*(5>3) UND (7.3>-1.2) => true */
(((a>=b)||(f<=g)) && i) /* i entspricht i!=0)*/
```

- Zuweisung =

```
links = <Ausdruck>;
```

  - legt ausgewerteten Ausdruck in Variable links
  - kann auch als boolsches Resultat verwendet werden

```
if (i=0) printf("ich werde nie gedruckt");
```

- Arithmetische Zuweisung: `+=, -=, *=, /=, %=`

```
<variable> op= <ausdruck>;
```

  - entspricht `<variable> = <variable> op <ausdruck>;`
  - natürlich auch mit Ergebnis

- Inkrement: `++, --`
  - addiert bzw subtrahiert 1
  - liefert Inhalt von j vor dem Inkrement: `j++`
  - liefert Wert von j nach dem Inkrement: `++j`
  - liefert auch Ergebnis: `do … while (j--)`

- Klammerregeln
  - Klammer hat höchsten Vorrang
  - gruppiert Formelteile

```
(a+b)*c != a+b*c
1 + ((3+1)/(8-4)-5)
```

- Vorrang-Regeln (precedence)

Typ	Assoziativität
Klammern	links -> rechts
unäre Ops	rechts -> links
Multiplikation, Addition	links -> rechts
relational	links -> rechts
bitweises UND, ODER	links -> rechts
logisches UND, ODER	links -> rechts
Zuweisung	rechts -> links

- Wirkung auf verschiedene Typen

```
int j=5;
float f=5;
einint = j / 2;     /* einint  == 2 */
einfloat = f/2;     /* einfloat == 2.5 */
```

  - Typ des Ausdrucks wird durch Elemente des Ausdrucks bestimmt

```
einfloat = j/2;     /* einfloat == 2.0 */
```

- Elegant

```
aktuelles_zeichen = name[index++];
summe += element;
```

- Etwas kryptisch:

```
a = b = c;
if (a=b) …;
```

- Bitte nicht:

```
x = j * j++;
einfloat = einint = 3.5;
einint = einfloat = 3.5;
```

- Anweisungen (Statements)
  - `<Ausdruck>;`
  - Selektion, Iteration, Sprung (siehe 2.3.1.3)
  - Increment
  - mehrere geklammerte Statements

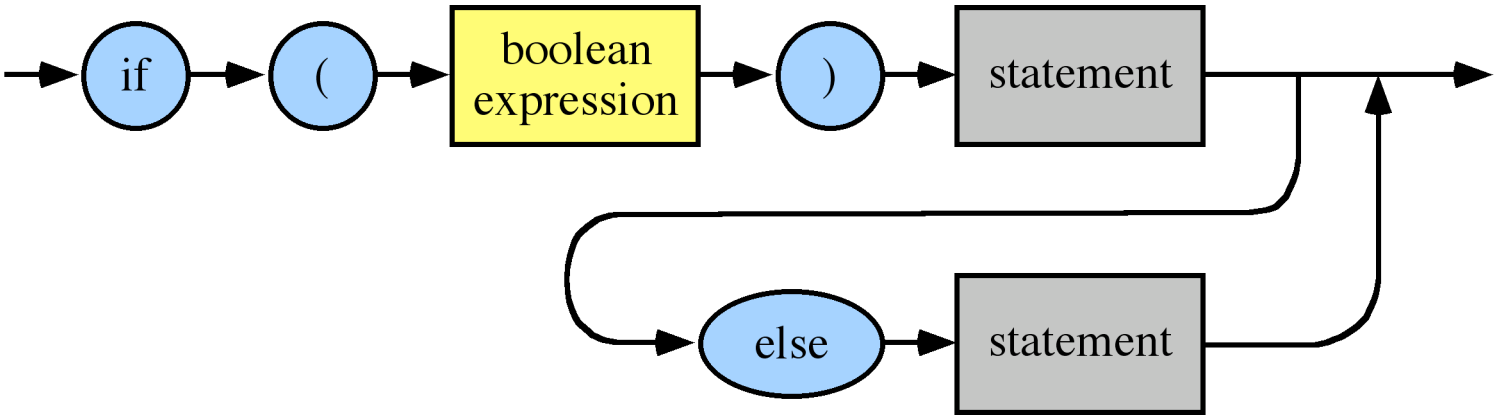
```
{
a=7;
anz_gedr_zchn=printf("auch printf hat Ergebnis");
gesZeichen += anz_gedr_zchn;
if (gesZeichen > 80) printf("\n");
}
```

- {} macht aus mehreren Statements ein Statement (Block)
  - {…;…;…;}
  - macht Syntax-Beschreibung einfacher
  - Variablenvereinbarung möglich

```
if (a>b) <statement>;
else <statement>;
```

2.3.1.3 Kontrollfluß

- Verzweigung
  - then implizit, immer vorhanden
  - else optional



- Beispiele

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

if (b>a) /* Werte tauschen */
   { swap = a; a = b; b = swap; }

if (divisor==0) printf("Fehler\n");
else quotient=dividend/divisor;
```

- Boolean Expression
  - eigentlich normaler Ausdruck mit Zahl als Resultat
  - true falls Resultat ≠ 0!

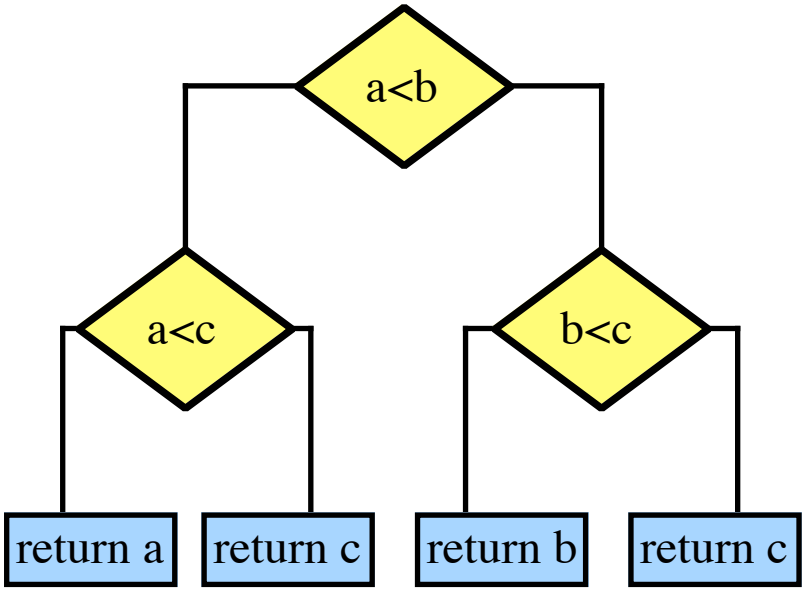
```
if (a-b) printf("a ist ungleich b");
```

  - klassischer Fehler: a=0

```
if (a=0) …;   /* immer falsch */
if (b=5) …;   /* immer wahr */
```

- geschachtelte Verzweigung

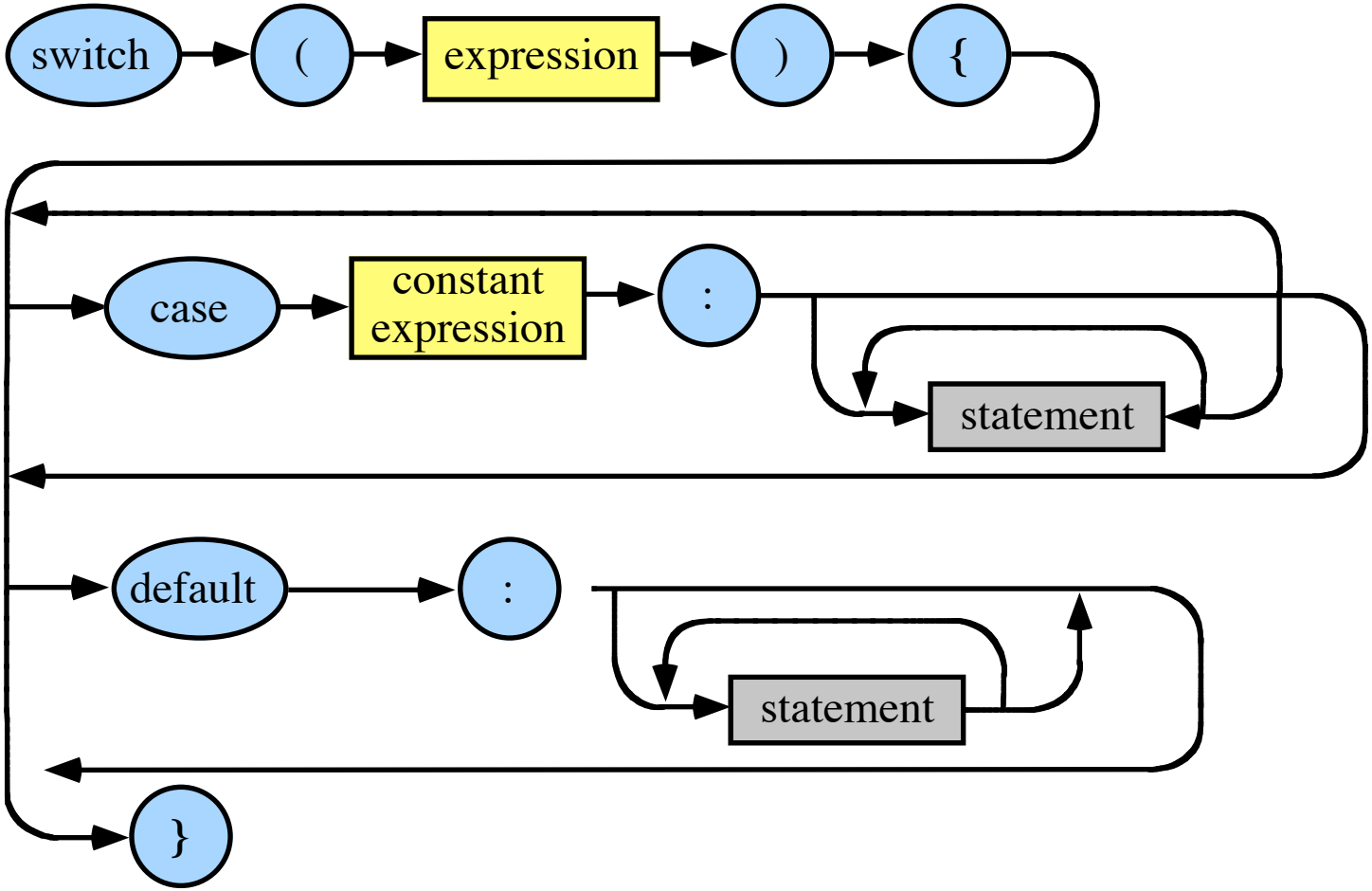
```
int min(a,b,c)
int a,b,c;
{  if (a<b)
      if (a<c) return a;
      else return c;
   else if (b<c)
      return b;
      else return c;
}
```



- Kaskadierte Verzweigung
  - mehr als 2 Fälle
  - nacheinander abfragen

```
int fallunterscheidung (eingabe)
char eingabe;
{
   if (eingabe == 'A')
      return 1;
   else if (eingabe == 'B')
      return 2;
   else if (eingabe == 'C')
      return 3;
   else if (eingabe == 'D')
      return 4;
   else
      return -1;
}
```

- Mehrfach-Verzweigung



  - expression wird ausgewertet
  - Resultat wird mit constant expressions verglichen
  - Ausführung wird bei 'true'-Zweig fortgesetzt

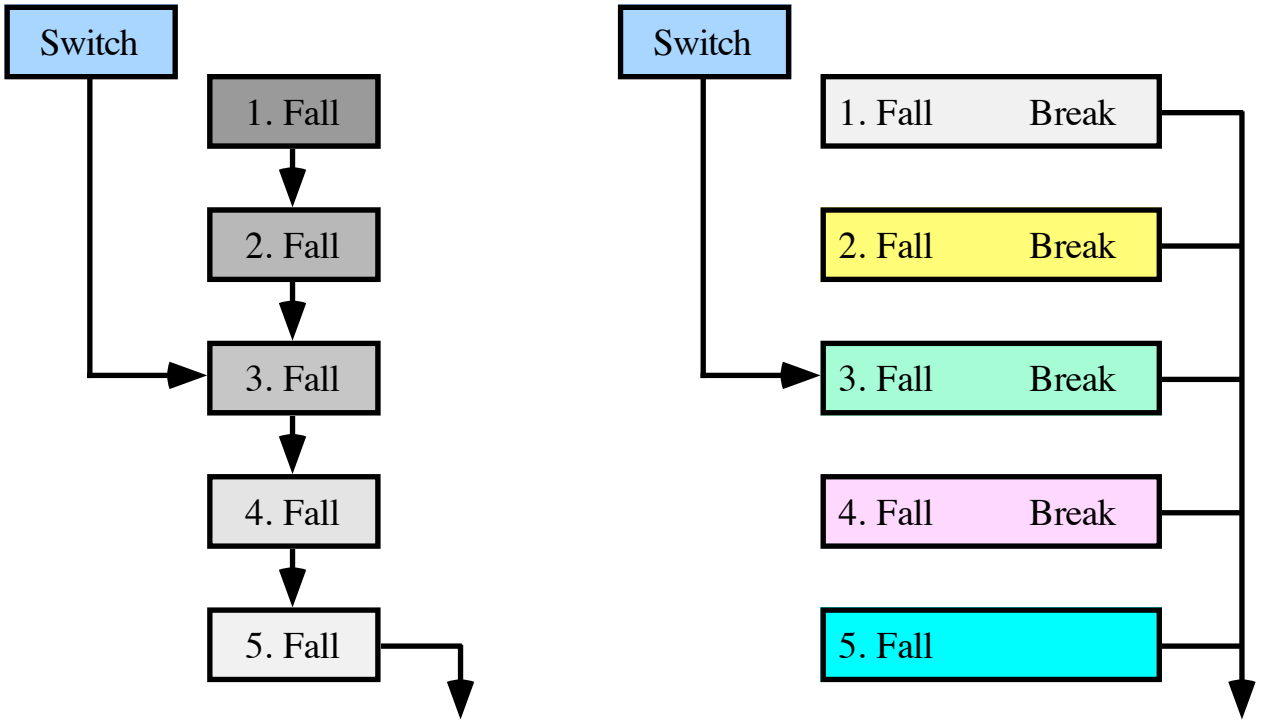
- Beispiel: Rechnen

```
double evaluate(double op1,char operator,double op2)
{  switch (operator)
   {  case '+':  return op1 + op2;
      case '-':  return op1 - op2;
      case '*':  return op1 * op2;
      case '/':  return op1 / op2;
      default: printf("Aetsch, Fehler!!!");
} return 0; }
```

- schlecht: alle folgenden Statements werden auch ausgeführt

```
double evaluate(double op1,char operator,double op2)
{  double erg=0;
   switch (operator)
   {  case '+':  erg = op1 + op2;
      case '-':  erg = op1 - op2;
      case '*':  erg = op1 * op2;
      case '/':  erg = op1 / op2;
      default: printf("Aetsch, Fehler!!!");
} return erg ; /* ist immer op1/op2 oder 0 */}
```

- break ist wichtiges Element von switch
  - verhindert Ausführung der restlichen Fälle




```
switch (eingabe)
{  case 'A':  ausfuehren(); break;
   case 'B': beenden(); break;
   case 'D': drucken(); break;
   default : printf("Fehlerhafte Eingabe");
}
```

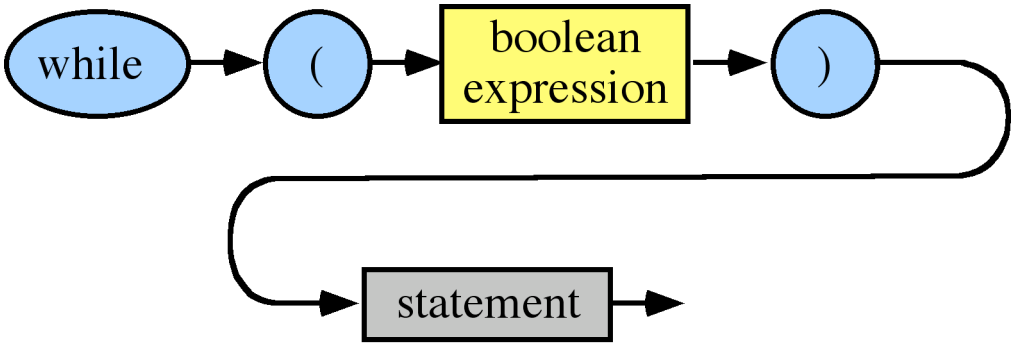
- Verbessertes Beispiel

```
double evaluate(double op1,char operator,double op2)
{  double erg=0;
   switch (operator)
   {  case '+':  erg = op1 + op2; break;
      case '-':  erg = op1 - op2; break;
      case '*':  erg = op1 * op2; break;
      case '/':  erg = op1 / op2; break;
      default: erg = 0; printf("Aetsch, Fehler!!!");
} }
```

- Beispiel: einfaches Menue

```
printf("Bitte waehlen Sie:\n S = Sichern\n \
            L = Laden\n N = Neu Anlegen");
scanf("%c",&eingabezeichen);
switch (eingabezeichen)
{  case 'S': case 's': savefile(); break;
   case 'L': case 'l': loadfile(); break;
   case 'N': case 'n': newfile();
}
```

- Schleife



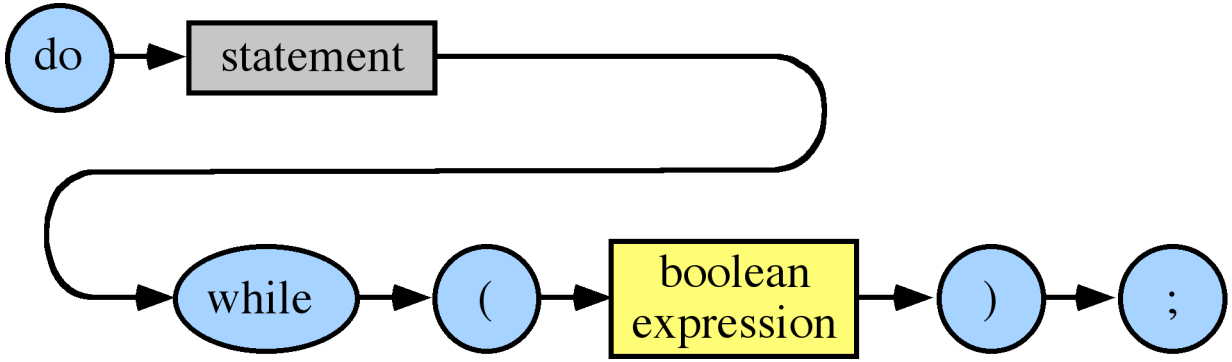
  - Bedingung wird am Anfang und nach jedem Durchlauf geprüft
- Beispiel n! (Fakultät, 2*3*4*5*…*(n-1)*n)

```
fak = 1; i = 1;
while (i <= n)
   {fak = fak * i; i = i + 1;}
```

- Beispiel: Leerstellen in der Eingabe zählen

```
printf("Satz eingeben: \n");
ch = getchar();
while (ch != '\n')
{  if (ch ==' ') num_of_spaces++;
   ch = getchar();}
printf("Anzahl Leertellen: %d",num_of_spaces);
```

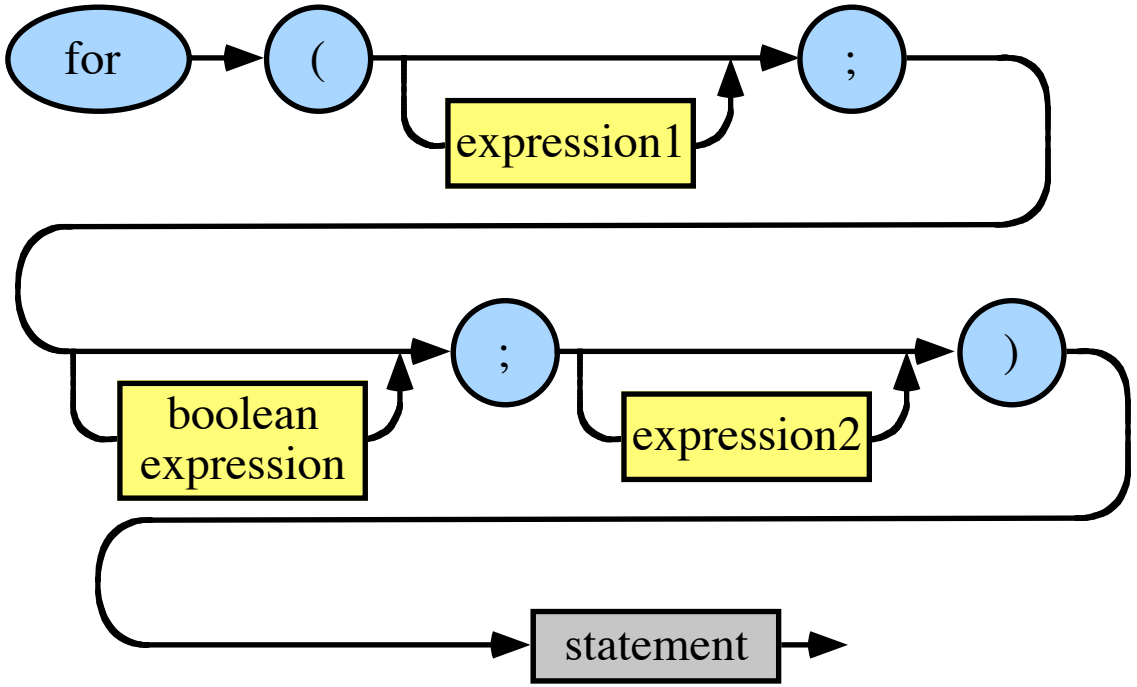
- Schleife mit Test am Ende
  - erst ausführen, dann testen
  => 'statement' wird mindestens einmal ausgeführt



- Beispiel: nochmal n!

```
fak = 1; i = 1;
do
   {fak *= i; i ++;
   } while (i <= n);
```

- Zählschleife
  - expression1 initialisiert Schleifenzähler und andere Variable
  - boolean expression entscheidet über Ausführung
  - expression2 wird nach der Ausführung von Statement ausgeführt
  - sollte Schleifenzähler inkrementieren



- Beispiel: schon wieder n!

```
fak = 1;
for (i = 1; i <= n; i+=1) fak = fak * i;
/* oder */
for (fak=1,i = 1; i <= n; ) fak = fak * i++;
```

- 3 Expressions bieten viele Möglichkeiten

```
for (fak = i = 1; n-i++; fak *= i );
for (sum = i = 0; n-i++; sum += i );
/* hacker's paradise */
```

- Vorsicht mit der Anzahl

```
for (i=0; i<n; i++) /* n-mal ausgeführt*/
for (i=0; i<=n; i++) /* (n+1)-mal ausgeführt*/
```

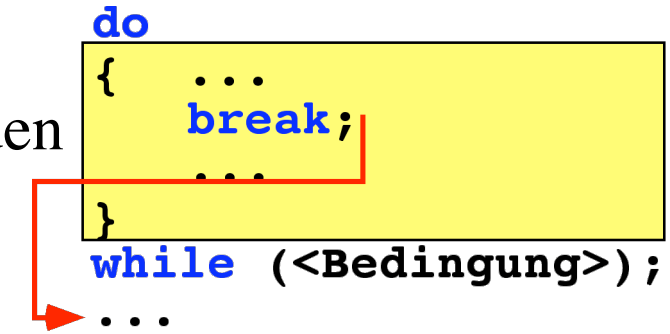
- Leerzeichen überspringen

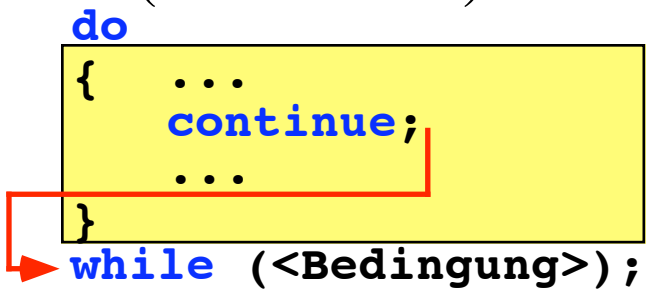
```
void skipspaces ()
{  int ch = ' '; /* kleiner Trick */
   for (; ch=' '; ch=getchar())
      ; /* Null Statement */
   ungetc (ch, stdin);
}
/* wollen wir nicht doch lieber while verwenden? */
```

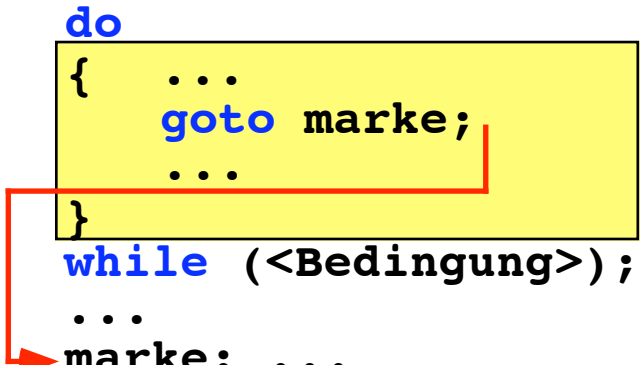
- Geschachtelte Schleifen
  - innere Schleife vollständig ausgeführt
  - mehrstufig

```
int main ()
{
   int j,k;
   printf("    1   2   3   4   5   6   7   8   9   10\n");
   printf("    ----------------------------------------\n");
   for (j=1; j <= 10; j++)
   {
      printf("%5d|",j);
      for (k=1; k<=10; k++)
         printf("%4d", j*k);
      printf("\n");
   }
   return 0;
}
```

- Sprünge: the good, the bad and the ugly
  - break, continue, goto <label>
  - direkte Einflussnahme auf den Kontrollfluss
  - Programm kann auch anders formuliert werden
- `break` bricht ab
  - umfassende Schleife
  - switch
  - Ausführung mit Statement nach Schleife/switch fortsetzen
  - immer nur die jeweils innerste Schleife (bzw. switch)
- `continue` setzt fort
  - bleibt in der Schleife
  - Schleifentest + evtl. nächste Iteration
- `goto` springt zu einer Marke
  - Marke definiert: `ich_bin_ein_marke: <statement>`
  - `goto ich_bin_ein_marke;`



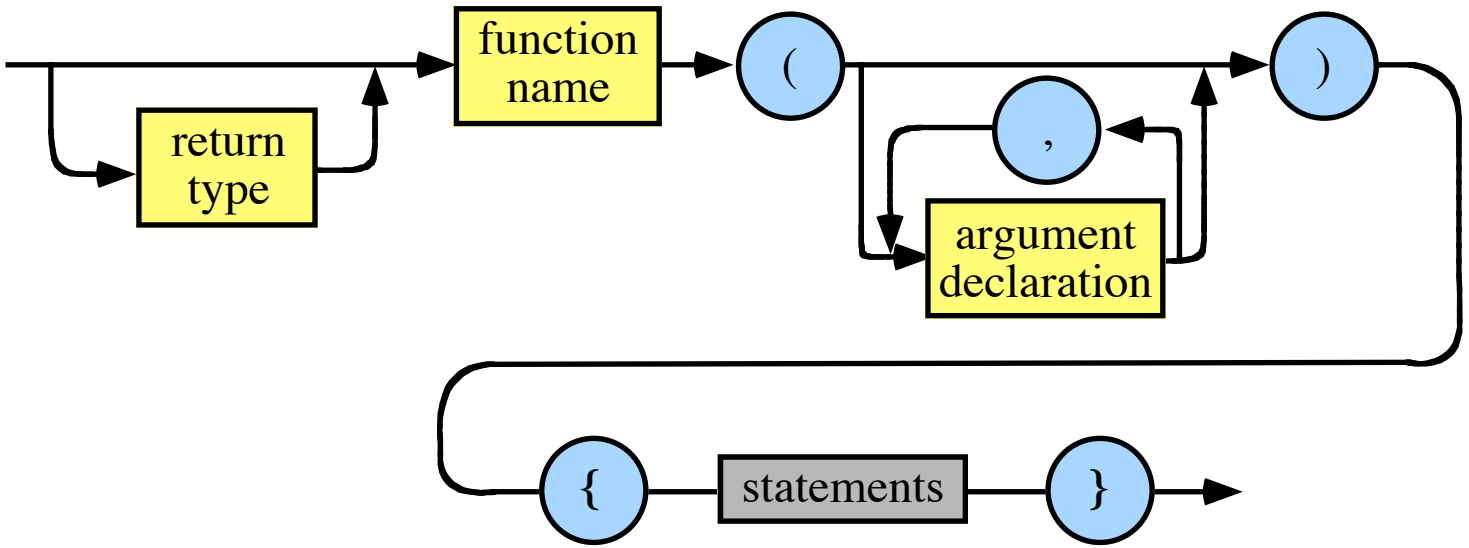




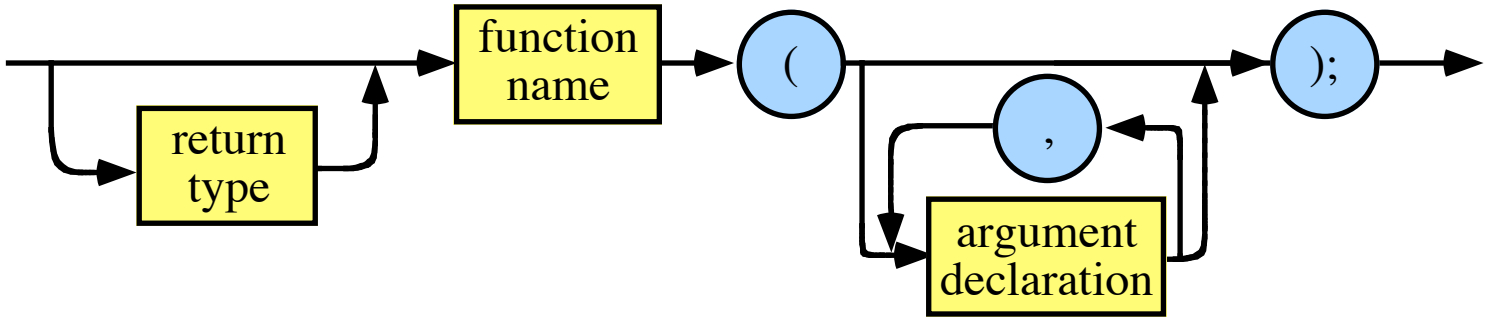
2.3.2 Funktionen im Detail

- Definierte Aufgaben gesondert programmieren
  - stepwise refinement
  - Problem zerlegen
  - häufig gebrauchte Komponenten
  - Werkzeugkasten
  - nur aktuelle Werte unterscheiden sich
  - Abstraktion
- Definition
  - Anzahl und Typ der Argumente
  - Statements im Rumpf der Prozedur

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



- Keine Funktionen in Funktionen
- Deklaration vor der ersten Verwendung
  - falls Funktion vor der Deklaration aufgerufen wird
  - oder in Header-Dateien

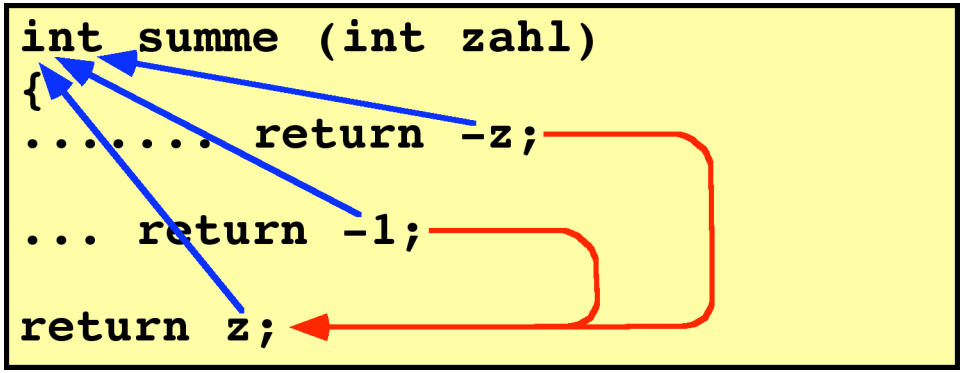



```
int f2 (int m, int n, int p); /* Deklaration */
...
int f1 (int a, int b)
{ ...
  f2(a,13,b+3);
  ...                    }
...
int f2 (int m, int n, int p) /* Definition */
{ ...
  f1(m,13);
  ...                    }
```

- Parameter
  - automatische Konvertierung ähnlich wie bei Zuweisung
  - es werden Werte übergeben
  - keine Variablen!
  - Werte werden in 'neue' Variablen gelegt
  - Trennung Wert-Variable
- Beispiel Werte-Parameter

```
void f(int argument)
{ argument = 3;}

int main ()
{  int a = 2;
   f(a);
   printf("%d\n",a);  /*wetten,dass 2 gedruckt wird?*/
   return 0;
}
```

- Rückgabewert
  - mit Funktionsresultat
  - return-Statement
  - Funktion wird sofort verlassen
  - mit Ergebnisrückgabe



- Resultate mit Seiteneffekt erzeugen
  - Parameter als Zeiger auf die Variablen
  - zeigen auf 'Behälter' für Rückgabewerte
- Beispiel: Vertauschen von 2 Variablen

```
void swap (int *x,int *y)
{  int temp;
   temp = *x;
   *x = *y;
   *y = temp;
}

main()
{  int a=2, b=3;
   swap(&a,&b);
   printf("a ist %d\t b ist %d\n,a,b);
}
```

- Siehe scanf()

```
int f (int z)
{
  z = 42;
};
```

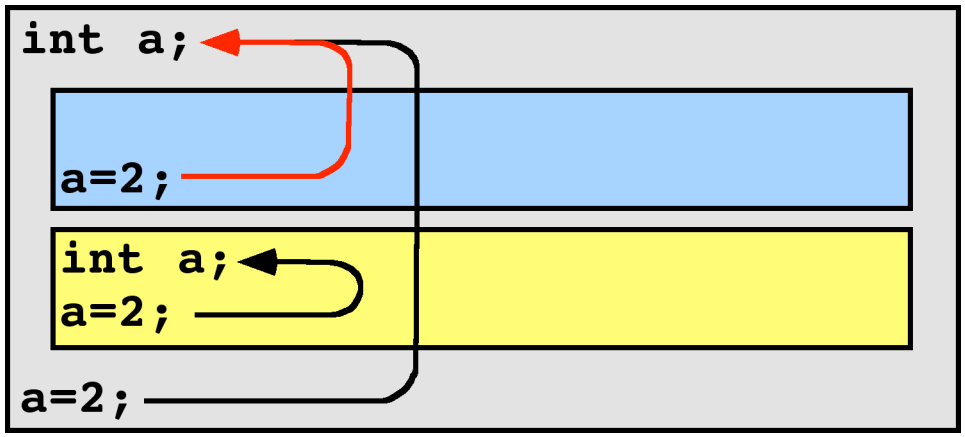
m | 42

```
int f (int *z)
{
   *z = 42
};
```

```
int main ()
{ int platz;
  ...
  f(&platz);
  ...
};
```

platz | 42

- Gültigkeit der Namen
  - Parameter und lokale Variablen
  - lokal gültig
  - überdecken weiter aussen definierte Namen



- void
  - Funktion ohne Resultat
  - Spezialtyp als Füllelement

```
void licht_an(int welches)
{…}
```

- Ein komplettes Programm

```
#include <stdio.h>

int summe (int zahl)
{  if (zahl >0) return zahl + summe(zahl-1);
   else return 0;
}
void allesummen (int max)
{ int lauf = 0;
   for (lauf = 1; lauf <= max; lauf++)
      printf("%3d: %6d\n",lauf,summe(lauf));
}
int einlesen()
{ int zahl;
   printf("Bitte Obergrenze eingeben:");
   scanf("%d",&zahl);
   return zahl;
}
int main ()
{  allesummen(einlesen()); return 0;}
```

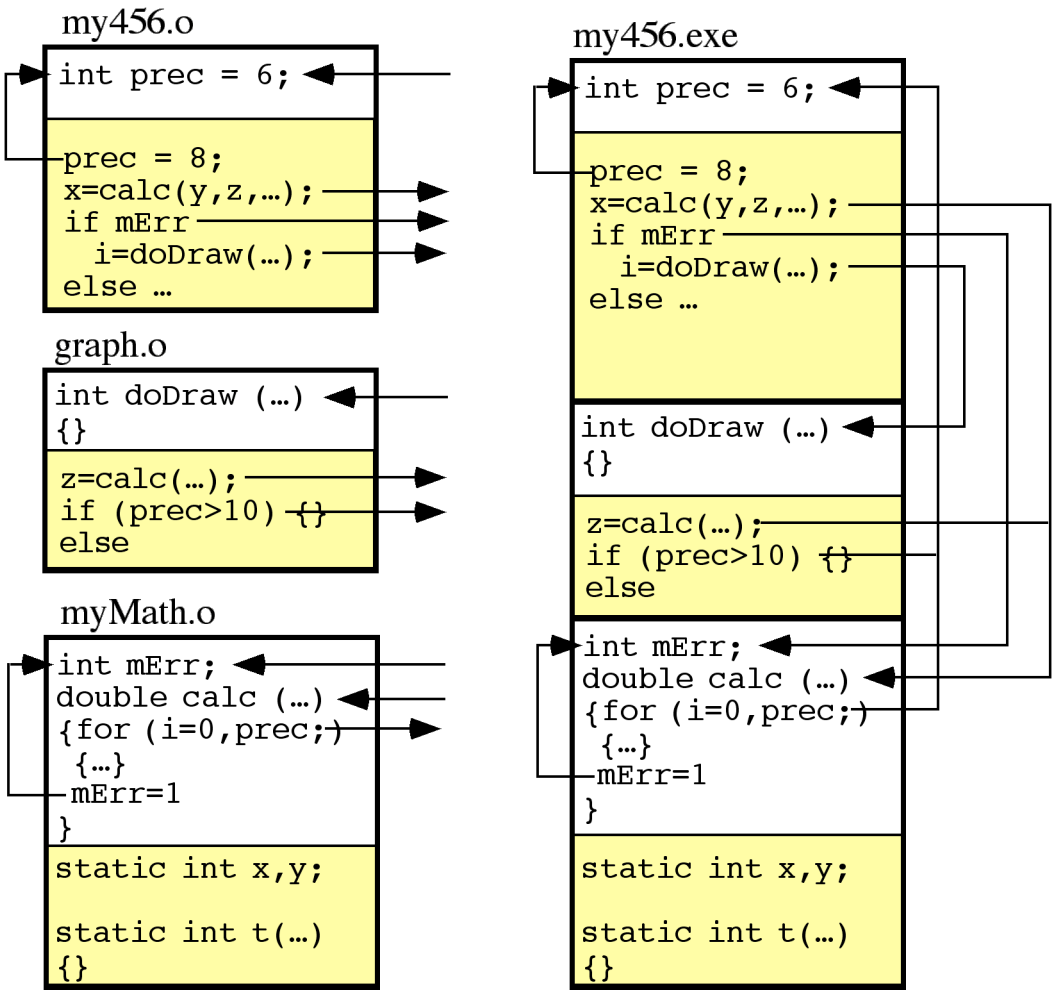
2.3.3 Module

- Programme auf mehrere Dateien verteilen
  - Strukturierung des Problemes
  - Gruppenarbeit
  - getrennte Übersetzung
- Schnittstelle definiert
  - Leistungen des Modules
  - nach außen sichtbar
  - Typen und Variablen
  - Funktionen
  - Implementierung und Details versteckt
- Bekanntgabe an andere Module
  - Datei `regler.h` enthält Schnittstelle
  - Datei `regler.c` enthält Implementierung
  - Import mit include-Pseudobefehl

  ```
  #include <modul.h>
  ```
- Übersetzung von Modulen
  - Parameter überprüfen,
  - Modul-Code und Referenzlisten erzeugen

- Zusammensetzen (Linken)
  - Zusammenkopieren der Module
  - Auflösen externer Referenzen: Einsetzen von Speicheradressen

Compiler Linker



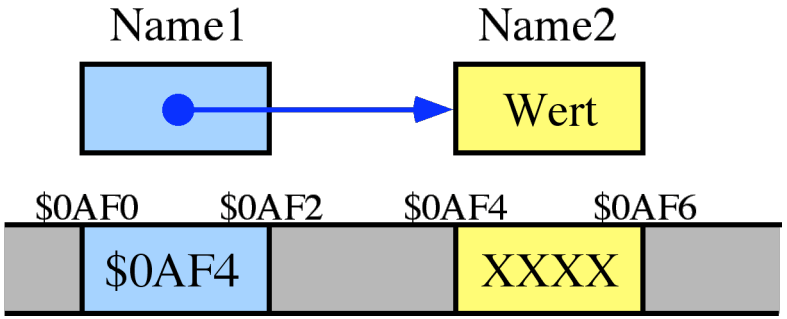
- Laufzeitumgebung: besondere Module
  - kommen mit dem Compiler
  - gehören oft zum Betriebssystem
  - standardisiert: ANSI, ISO, POSIX, …
- Vorgefertigte Funktionen
  - Standardwerkzeuge
  - Mathematische Funktionen
  - Ein/Ausgabe
  - String-Verarbeitung
  - Datum und Zeit
- Mathematische Funktionen: math.h

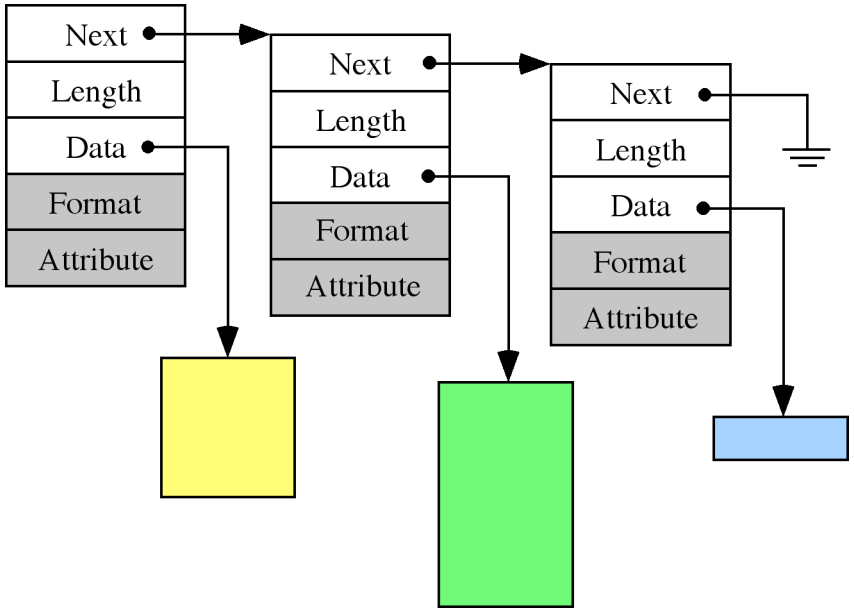
  sqrt(), log(), exp(), sin(), cos (), …
- Ein/Ausgabe: stdio.h
  - scanf(), printf(), getchar()
  - Datentyp FILE => Dateien
- String-Verarbeitung
  - siehe Kapitel 2.3.4

2.3.4 Arrays und Pointer

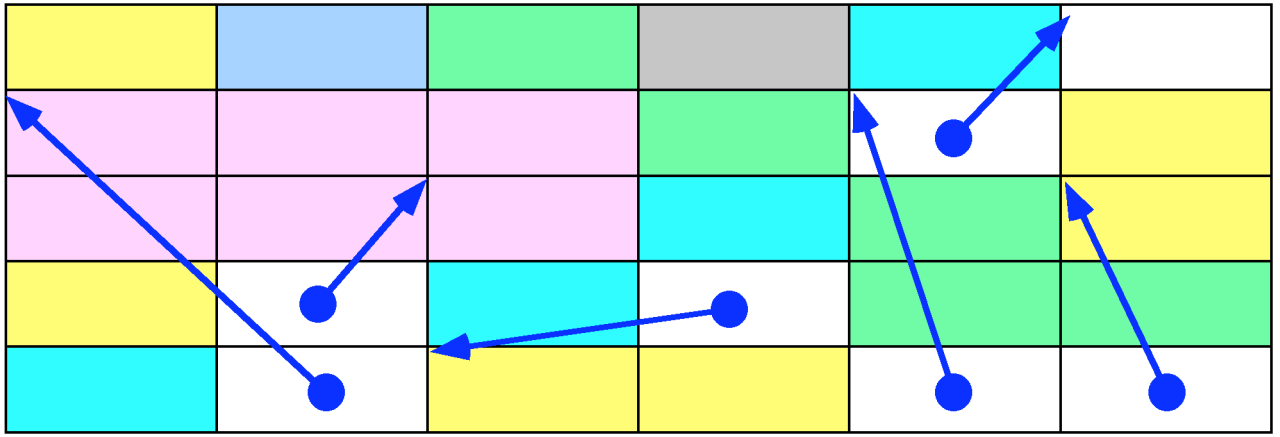
2.3.4.1 Zeiger (Pointer)

- Namen
  - identifizieren Objekte
  - von Menschen benutzbar
  - DNS: www.froitzheim.com
- Adressen
  - von Maschinen manipulierbar
  - schwer von Menschen verständlich
  - IP-Nummer: 134.60.77.64
  - 415-653-9516
- Referenzen
  - Verweis (Hinweis) auf andere Elemente
  - Bsp: Stellvertreter, Anrufumleitung
  - Hypertext-Referenzen
  - Listen, Bäume, …
  - Strukturinformation im Textprogramm
  - Alias/Verknüpfung (z.B. Windows 95, 98)





- Zeigervariablen
  - zeigen auf beliebige Datenobjekte

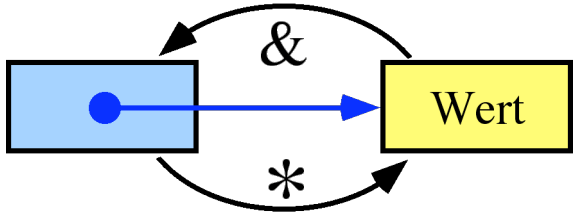


- '*' in Deklarationen und Definitionen

```
int *p; /* kein Speicherplatz für int! */
char ach, bch, *pch;
…
p = (int*)98;    /* Wo mag das wohl sein? */
p = p+1;         /* nun zeigt p auf Speicher 100 */
                 /* oder auf 102? */
                 /* Antwort: 98 + sizeof(int) */
```

- Address-of Operator '&'

```
pch = &ach;
scanf("%f",&messwert)
```



- Dereferenzierung mit '*'

```
pch = (char*)100;
printf("Speicherzelle 100: %d",*p);
```

- Bedeutung von '*' entgegengesetzt in:
  - Definition / Deklaration: 'pointer-to'
  - Expression: 'content-of'
- Leerer Zeiger == NULL

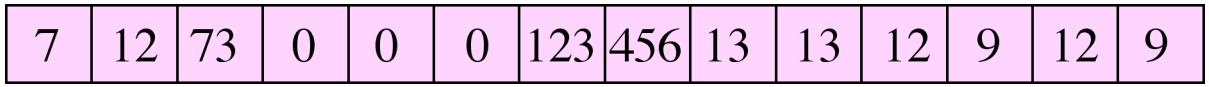
```
if (next == NULL) printf("Fertig");
```

- Pointer als Parameter
  - ermöglichen Rückgabe von Resultaten

```
int raten(int *wert, int min, int max)
{ int try = 0;
   do {  printf("\nBitte Eingabe: ");
         scanf("%d", wert);
         try = try + 1;}
   while (*wert < min || *wert > max);
   return try;
}
int eingabe;
/* Aufruf */
punkte = punkte + raten(&eingabe,17,99);
printf("der Kandidat hat %d geraten", eingabe);
```

- Übergabe großer Datenmengen ohne Kopie
  - Kopien kosten Zeit und Platz
  - aber: Veränderung in der Funktion evtl. unerwünscht

2.3.4.2 Arrays



- Arrays sind Vektoren, Matrizen, …
- Definition

```
<typ> <name> "[" <dimension> "]"
            {"[" <dimension> "]"};
```

  - Beginn immer bei 0
  - Dimension ist Längenangabe

```
int vektor [100];
float matrix [zeilen][spalten];
```

- Initialisierung

```
int rot [3] = {255,0,0};
char name [32] = "Hallo"; /* 32 byte */
char name2 [] = "Hallo";  /* 6 byte */
```

- Verwendung
  - meist elementweise

```
vektor[index]
matrix[zeilenindex][spaltenindex]
```

- Dualität Array - Pointer
  - Array-Ident zeigt auf erstes Element
  - zunächst das Erste (array[0])
  - abgekürzte Schreibweise

```
vektor[0]     entspricht:   *vektor
&vektor[0]    entspricht:   vektor

vektor + <expr> entspricht: &vektor[<expr>]
*(vektor + <expr>) entspricht: vektor[<expr>]

vektor + index
&vektor[5]-2

/* aequivalent */
   matrix[i][j]
   *(matrix[i] + j)
   *(*(matrix + i) + j)
```

  - Länge des Grundtyps wird beachtet

- Als Parameter für Prozeduren
  - Prozedurdefinition ohne Wissen über Länge
  - Länge fest oder als Parameter

```
void feldlesen(char *dasFeld, int feldlength)
{  int index;
   for (index=0; index < feldlength; index++)
      {printf("Element %d eingeben: ",index);
       scanf("%c", &dasFeld[index]);}
}

char einVektor[100];

int main()
{
   feldlesen(einVektor,100);
   drucken(einVektor,100); /* noch programmieren */
   return 0;
}
```

- Strings
  - char-Array

```
char name[32];
```

- Adressierung wie bei Arrays
  - Ende des Strings durch '\0' ('null terminated')
- Stringkonstanten in "…"

```
*"Konrad"           entspricht 'K'
*"Konrad"+2         entspricht 'M'
*("Konrad"+2)       entspricht 'n'
"Konrad"[6]         entspricht '\0'
"Konrad"            /* Adresse */
```

- Zuweisungen nur elementweise

```
name[0]='A'; name[1]='l'; name[2]='f';
name[3]='\0';
printf("%s\n",name);
```

```
...
Alf
```

- Funktionen zur Stringverarbeitung
  - in string.h
- strncpy (strcpy gefährlich …)

```
char name [32];
…
printf("%s",name);
strncpy(name, "Jeremias Jammermeier", sizeof(name)-1);
printf("%s\n",name+8);
```

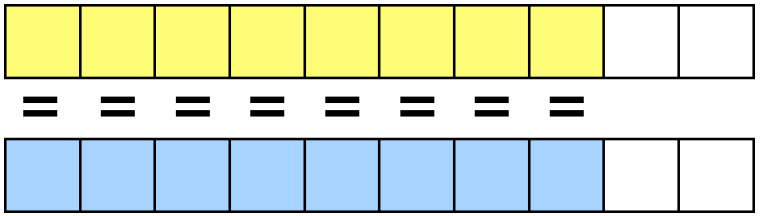
```
...
Alf Jammermeier
```

- strlen, strcat, strcmp, …

- Idee für strlen

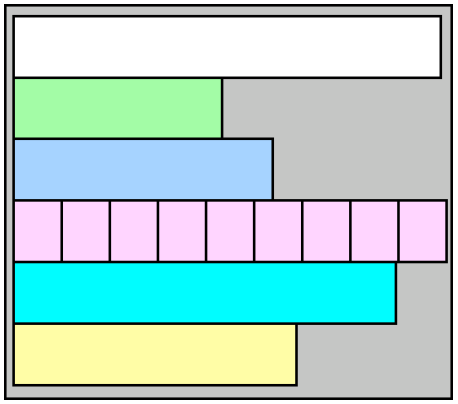
```
int strlen(char *theStr)
{ int length = 0;
   while (*theStr != '\0') /* 0 reicht auch */
      {length++; theStr++;}
   return length;
}
```

- Idee für strcmp



```
int strcmp(char *aStr, char *bStr)
{
   if (strlen(aStr)!=strlen(bStr)) return 0;
   while ((*aStr == *bStr) && *aStr)
      {aStr++; bStr++};
   if (*aStr=='\0') return 1;
   else return 0;
}
```

2.3.5 Structures



- C-Form von Records
  - Objektbeschreibung, Personaldaten
  - heterogene Datenelemente
  - besonders in Datenbanken
  - verschiedene Felder
  - evtl. Zeiger
- Typ struct
- Strukturdeklaration

```
struct Wagenbeschr
   {  char name[32];
      enum helptype {Schlaf,Abteil, …} Wagentyp;
      int Baujahr;
      struct Wagenbeschr *next;
   };
```

- Variablendefinition

```
struct Wagenbeschr ShellWagen, BPWagen, Aralwagen;
```

- Zuweisung

```
strncpy(ShellWagen.name,"2-17 85003201",31);
ShellWagen.Wagentyp = Kessel;
ShellWagen.Baujahr = 1972;
ShellWagen.next = NULL;
Aralwagen = {Kessel, 1988, NULL};

BPWagen = Shellwagen;
```

- Vergleich nicht als Einheit

```
if (BPWagen == Shellwagen) /* falsch */
/* illegal structure operation */
```

- typedef macht eine bestimmte structure zum Typ

```
typedef struct Wagenbeschr WagenType;

WagenType EssoWagen;

/* natuerlich auch fuer Standardtypen */

typedef double Gewicht;
Gewicht maximal_zulaessiges_Gesamtgewicht;
```

- Initialisierung

```
Wagenbeschr SpeiWa = {Speise, 1978, NULL};
Wagenbeschr PersWa = {.Wagentyp = Personen,
                      .baujahr =1979}; /* rest 0 */
```

- Funktionsaufruf
  - {} Auf Parameterposition möglich
  - mit Typecast

```
typedef struct {char *name; int number} X;

void fn1(const X *x);
{…;}

fn1(&a); /* benannte Variable */
fn1(&(X){"name", 42}); /* anonyme Variable */
fn1(&(X){.name="name", .number=42});
fn1(&(X){"name", random()});
fn1(&(const X){.name="name", .number=42});
        /* anonyme Konstante */
```

- Pointer auf structures

```
WagenType *wagenPtr;
wagenPtr = &EssoWagen;
*wagenPtr = BPWagen;
```

- Zugriff auf Komponenten
  - Problem: '.' bindet stärker als '*'

```
/* falsch: printf("%s",*wagenPtr.name) */
(*wagenPtr).Baujahr = 1990;
```

  - neuer Operator:

```
wagenPtr -> Baujahr = 1990;
printf("%s",wagenPtr -> name);
```

- Parameter und Resultat von Funktionen

```
WagenType *newcopy (Wagentype *derWagen)
{ WagenType *help;
   help = malloc(sizeof(Wagentype));
   if (help == NULL) printf("%s","Out of memory");
   else *help = *derWagen;
   return help;
}
```

2.4 Systematische Fehlersuche

- Software hat Fehler
  - komplexe logische Systeme
  - Programme mit 1 Million Zeilen und mehr
  - Interaktion mit anderen Programmen
  - unscharfe Eingabe
- Verifikation schwer evtl. unmöglich
  - unentscheidbare Probleme (=> Goedel)
  - vollständige Aufzählung der Randbedingungen
- Programm ordentlich aufschreiben
  - systematisch einrücken
  - aussagekräftige Namen
  - don't get fancy
- Programm oft ausführen
  - mit verschiedenen Parametern
  - auf verschiedenen Rechnern
  - andere Personen testen lassen
- "Debugging is great fun" [Anonymous]

```
while (x == y) {
        something();
        somethingelse();
}
finalthing();
```

- Typische Probleme
- Bedingungen
  - Test auf Gleichheit: (a <= 0) oft besser als (a == 0)
  - "echt größer" (>) oder "größer-gleich" (>=)?
- while-Schleife
  - Endebedingung falsch
  - Seiteneffekte auf Endebedingung
- Variable nicht vorbesetzt
  - häufig bei erratischen Fehlern

```
int  teiler;
scanf("%d",b);
a = b/teiler; /* Was steht wohl in teiler? */
```

  - Programm läuft nur einmal
  - Programm läuft nur nach bestimmten anderen Programmen
  - Programm läuft, wenn Variable eingefügt wird
- Pointer zeigt irgendwohin
  - NULL-Pointer, …
  - Speicher wird überschrieben

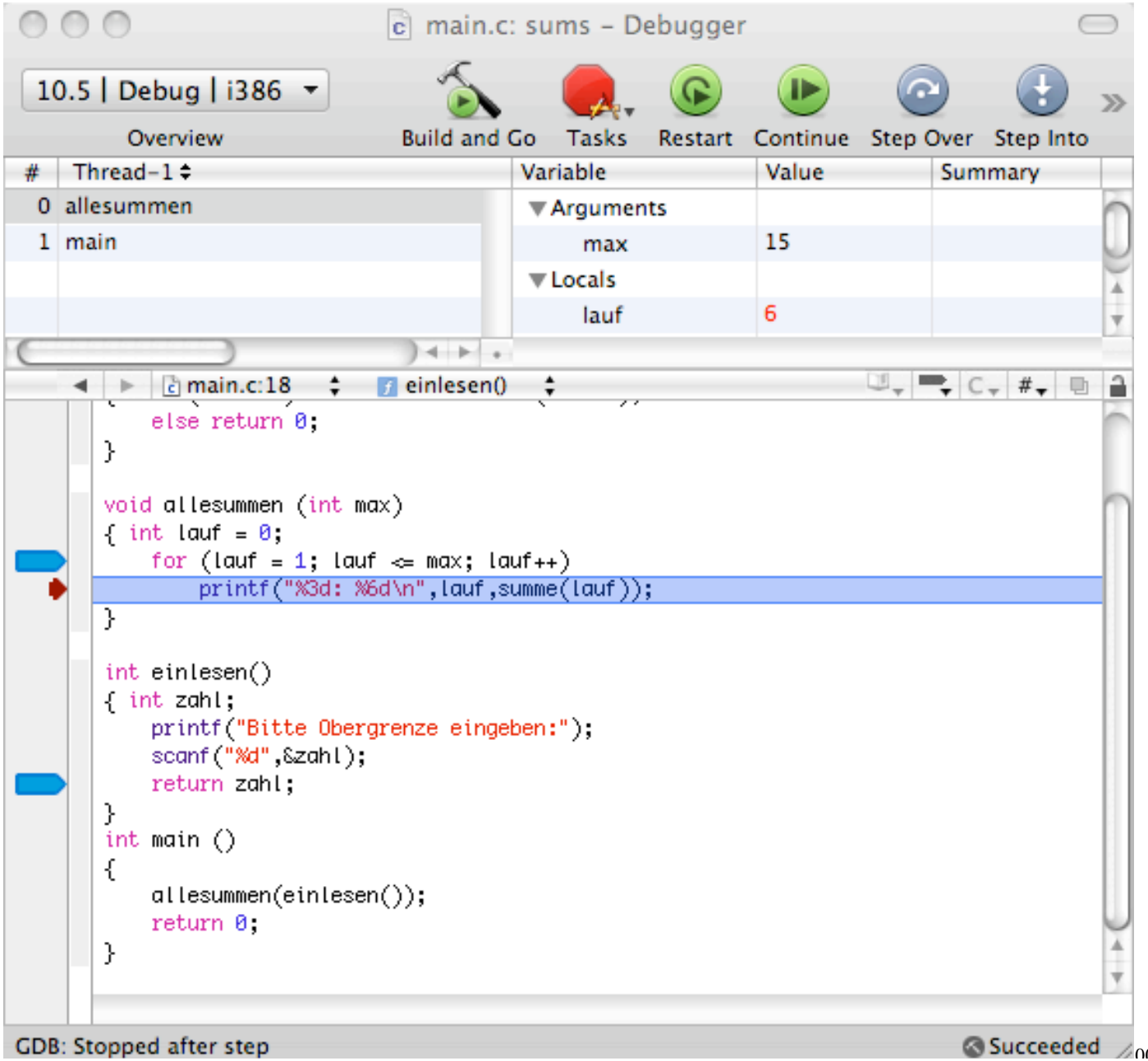
- Systematische Vorgehensweise
  - Beobachtung des Problemes (reproduzieren!)
  - Hypothesenbildung (warum?)
  - Messen (wenn - dann)
  - Beseitigen (programmieren)
- Wolfs-Zaun
  - Fehler eingrenzen
  - schrittweise eingrenzen mit printf()
- Haben die Variablen den richtigen Wert?
  - am Ende von Funktionen
  - nach jedem Statement
- Besondere Abfragen
  - Annahme: Variable sollte ≠ 0 sein

```
if (<variable>==0) printf("Annahme verletzt\n");
```

oder: `assert(<variable>!=0); /* aus assert.h */`

  - Fehlermeldung und evtl. Abbruch
  - assertion (Zusicherung)

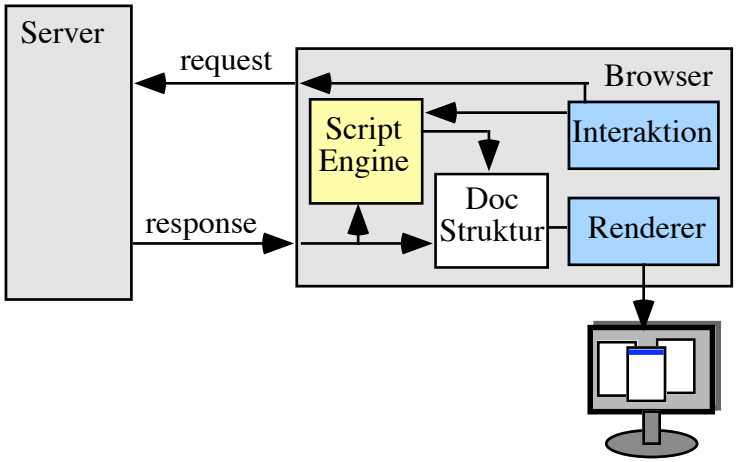
- Debugger
  - Variablen:Inspektion , Werte setzen
  - Aufrufkette
  - Breakpoints
  - trace
  - step

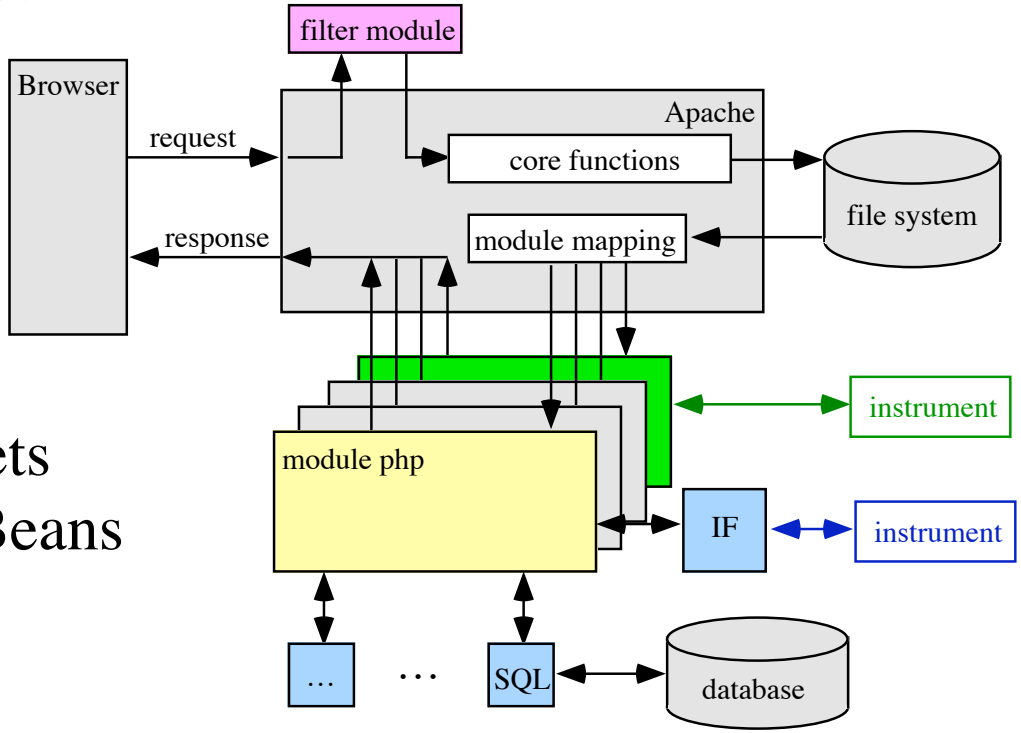


2.5 Reaktives Programmieren

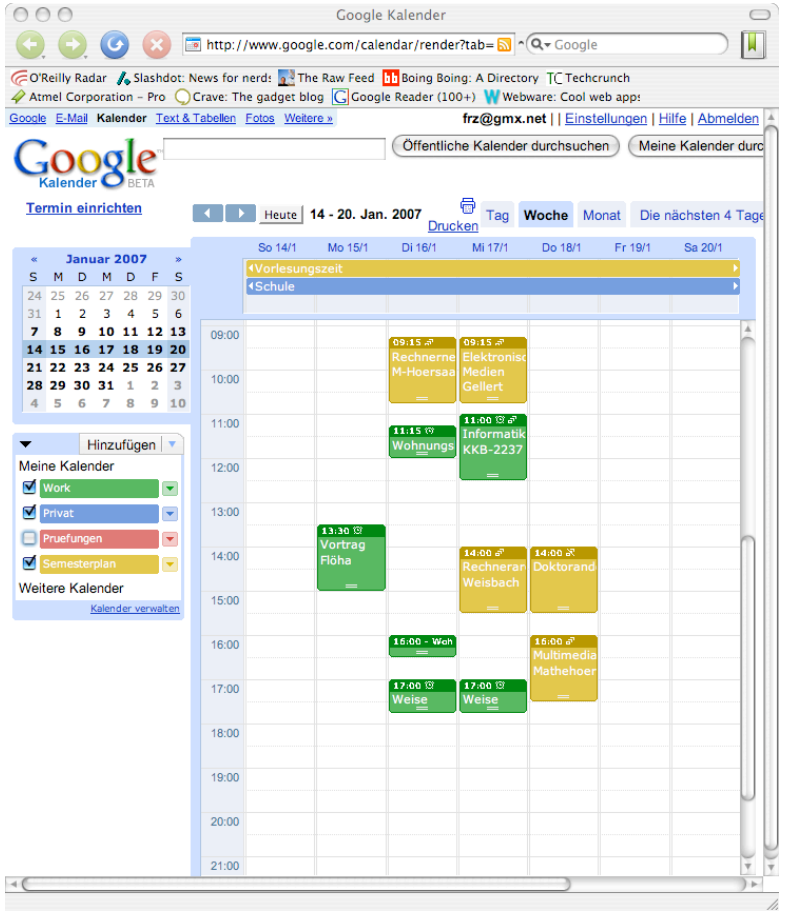
2.5.1 Ausführungsmodell WebApplikation

- Mikro-interaktiv
- Benutzeroberfläche
  - Browser
  - html mit Grafiken als Substrat
  - individuelles Verhalten: JavaScript
- Server
  - Datenbank
  - Geräte
  - Apache und Apache-Module
- Scripting
  - client-side: Javascript, Java Applets
  - server-side: ASP, php, Java und Beans
- Beispiele
  - http://rr.informatik.tu-freiberg.de
  - Shopping-Seiten



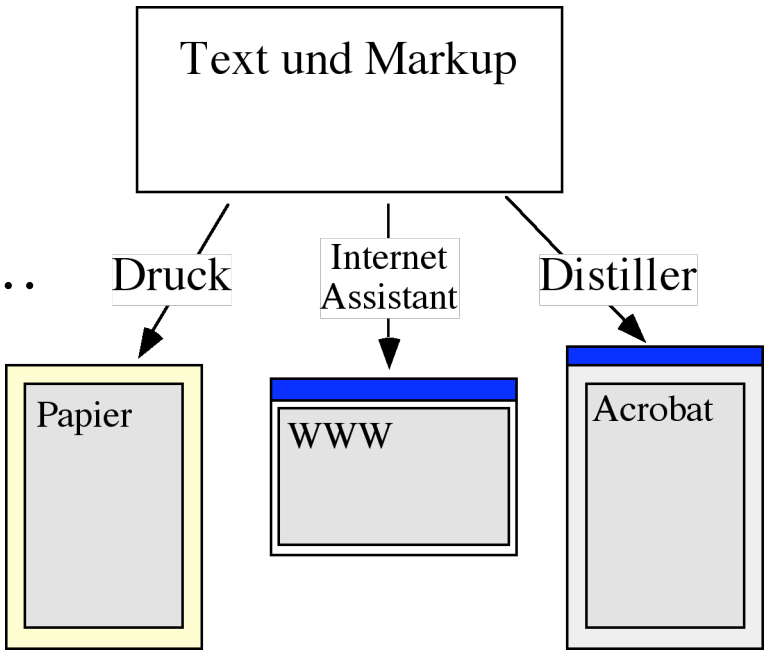


- Programmierparadigma
- Infrastruktur
  - Server:(Apache+php|JavaBeans)
  - Server:Datenbank
  - Client:(InternetExplorer|Firefox|Safari)
- Krümelware
  - kleine, vorgefertigte Bauteile (-> Objekte)
  - abgeleitete (erweiterte) Objekt
  - inkrementelles Programmieren
  - Einfügen und Erweitern statt Bauen
  - Gesamtkonzept? Architektur?
  - Datenfluß designen
- AJAX
  - Asynchronous JavaScript And XML
  - maps.google.com
  - flickr.com
  - docs.google.com, google calendar



2.5.2 HTML, die 'Sprache' des WWW

- Hypertext Markup Language
  - Berners-Lee 1989
  - Zweck: Verknüpfung von Dokumentation der Hochenergiephysik
  - keine Bilder - textbasierte Klienten
  - Standard Generalized Markup Language
  - Document Type Definition (DTD) von SGML
  - Hypertext-Referenzen: URLs eingebettet in das Dokument
  - http://www.w3.org/TR/REC-html40/
- Markup
  - logische Struktur für Text
  - Überschrift, normaler Paragraph, Zitat, …
  - Fußnote, Literaturverweis, Bildunterschrift, …
- Zuordnung der Attribute beim Satz
  - Autor produziert Inhalt und Struktur
  - Drucker setzt
  - Corporate Identity …
- HTML: ASCII-Text + <A>-Tag



- Beispieltext:

```
<HTML> <HEAD>
<TITLE> Ein HTML-Beispiel </TITLE>
</HEAD> <BODY>
<B>Dies</B> ist ein Hypertext Dokument.
<P>Mit einem Bild: <IMG SRC="bild.gif"> <BR> und
einem
<A HREF="Beispiel1.txt"> Hyperlink </A> </P>
```

Dies ist ein Hypertext Dokument.

Mit einem Bild:

und einem Hyperlink

```
</BODY> </HTML>
```

- Weitere Elemente siehe z.B. http://www.selfhtml.org/
  - Listen, Stile
  - Formatierung

2.5.3 JavaScript

- Programmfragmente in HTML
  - Verbesserung von HTML-Seiten auf der Klienten-Seite
  - Fenstergröße und -Gestaltung
  - Menus, Effekte, …
  - Beispiel: http://maps.google.com
- Interpreter im Browser
- Eingebettet in HTML
  - script-Tag

```
<html><head><title>Test</title>
<script language="JavaScript">
<!--
  alert("Hallo Welt!");
//-->
</script>
</head><body>
</body></html>
```

- Oder in anderen HTML-Tags

```
<html> <head>
   <title>JavaScript-Test</title>
   <script language="JavaScript">
   <!--
    function Quadrat(Zahl)
     {var Erg = Zahl * Zahl;
      alert("Quadrat von " + Zahl + " = " + Erg);  }
   //-->
   </script> </head>
   <body> <form>
   <input type=button value="Quadrat von 6 errechnen"
      onclick="Quadrat(6)">
   </form> </body>
   </html>
```

- Eventhandler
  - Attribut in html-Tags
  - beschreiben Ausführungsbedingung
  - Aufruf einer JavaScript-Funktion
  - onload, onclick, onmouseover, onkeydown, …

- Sprache
  - Notation ähnlich Java
- Anweisungen
  - Zuweisungen

```
zahl = 0; zahl++; zahl+=1;
```

  - Bedingte Anweisungen und Schleifen

```
if (Zahl<0) zahl = 0;
while (idx<100) {…; idx++}
for(i = 1; i <= 100; i++)
      {…}
```

  - Funktionsaufrufe

```
alert("Und hier ein kleiner Hinweis");
```

  - Klammern mit {}

```
if (Ergebnis > 100)
{ Ergebnis =0; Neustart(); }
```

- Variablen: kein ordentliches Typenkonzept
  - Vereinbarung mit 'var'
  - Typ Zahl oder String
  - wird bei der ersten Zuweisung festgelegt

```
var Antwort = 42;
var Frage = "The Question for god …"
```

  - global oder in Funktion lokal
- Objekte: Eigenschaften und Methoden
  - selbstdefiniert oder vordefiniert in Umgebung
  - window, document, images, links, array, …
  - Objekt erzeugen mit `var einobjekt = new <object>;`

```
var einFenster = window.open(<ref>,<titel>,<parms>);
```

  - Datenstruktur (Felder = Eigenschaften)

```
einFenster.name = "Lustig";
```

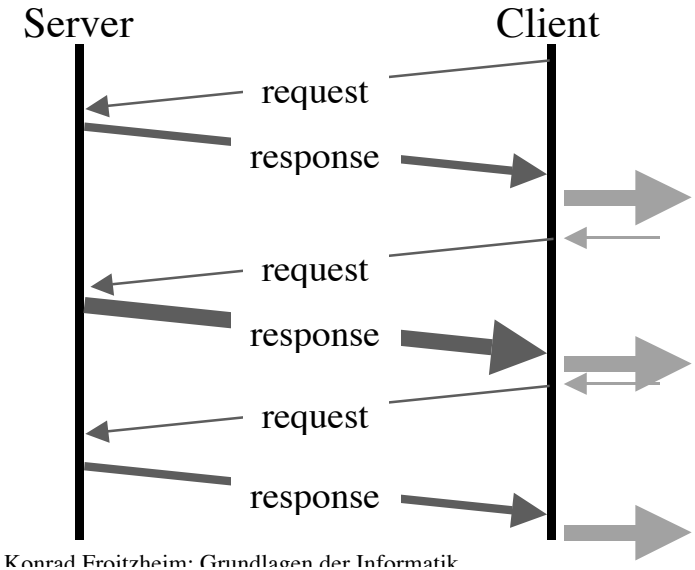
  - Methoden zur Manipulation: print, blur, moveTo, scrollBy, …

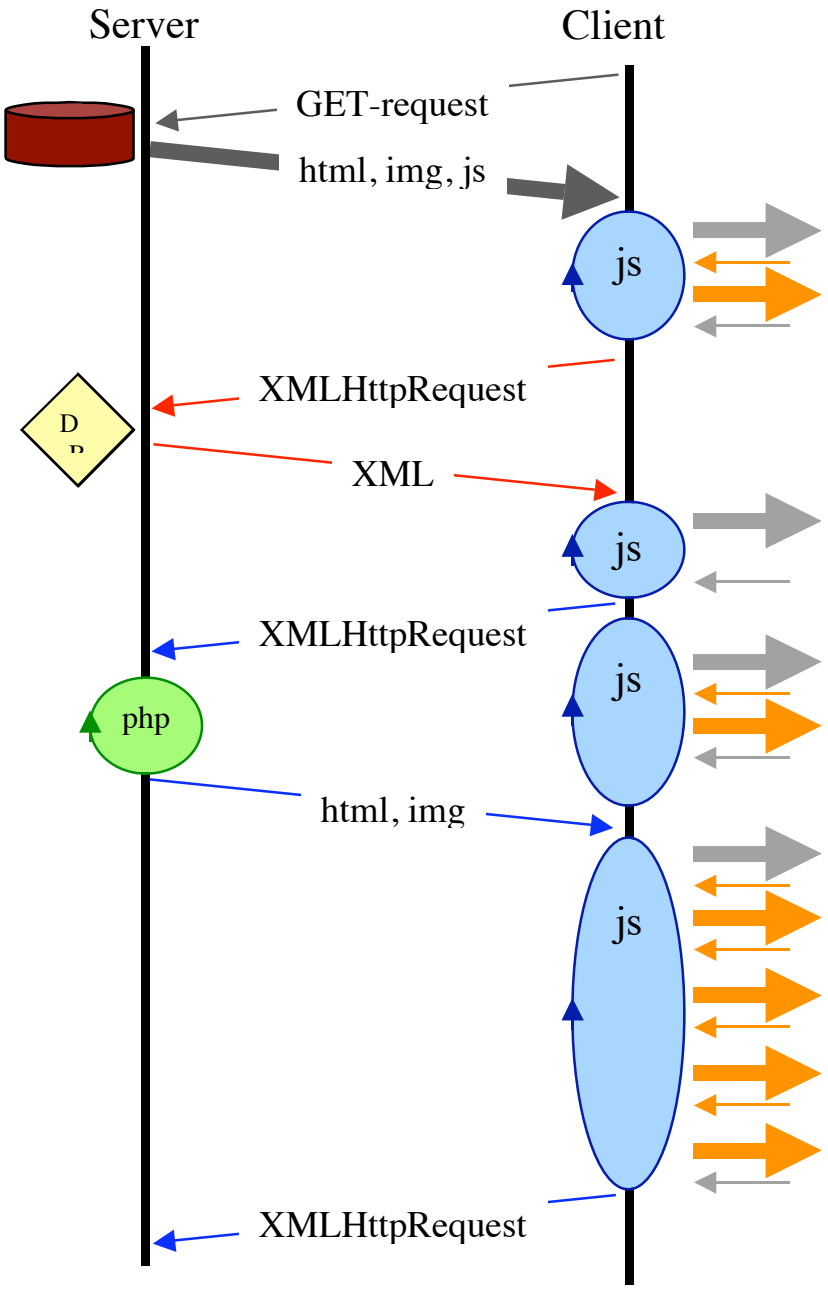
```
einFenster.resizeTo(500,400);
```

- Funktionen
  - Anweisungsblock mit Variablen
  - Aufruf aus anderen Funktionen und in Eventhandlern

2.5.4 AJAX

- Architektur von AJAX-Applikationen
- Programm im Browser (Client)
  - JavaScript
  - AJAX Libraries
- Server
  - Webserver, Datenbank
  - Serverside Scripte
- Leichtgewichtige Kommunikation
  - XMLHttpRequest
  - synchron und asynchron





- XMLhttpRequest
- JavaScript Klasse
  - erstmals in IE 5
  - Interface für Http
  - ohne Benutzerinteraktion
  - synchron und asynchron

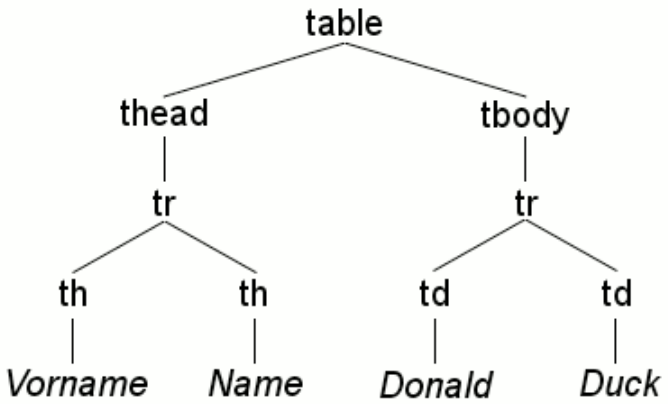
```
function createXMLHttpRequest() {
 try {return new ActiveXObject("Msxml2.XMLHTTP"); } catch(e) {}
 try {return new ActiveXObject("Microsoft.XMLHTTP"); } catch(e) {}
 try {return new XMLHttpRequest(); } catch(e) {}
 alert("XMLHttpRequest not supported");
 return null;}
```

- Properties
  - readystate, status
  - onreadystatechange
  - responseText
- Methoden
  - open
  - send

```
var xhReq = createXMLHttpRequest();
xhReq.open("get", "sumget.phtml?num1=10&num2=20", true);
xhReq.onreadystatechange = function() {
   if (xhReq.readyState != 4)  { return; }
   var serverResponse = xhReq.responseText;
   ...};
xhReq.send(null);
```

- Das X in AJAX
- Markup
  - Markup: Trennung Struktur - Inhalt
  - logische Struktur der Seite
  - Bsp: Überschriften, Absätze, Zitate, …
- XML: eXtensible Markup Language
  - Syntax für Markup
  - Semantik in XSL oder CSS
- Document Object Model DOM
  - baumartige Struktur der Dokumente
  - Zugriff auf Dokumenteninhalt (=Objekte)
  - Inhalt, Struktur, Stil
- AJAX
  - XML als ein Transfer-Format für Inhalt
  - Manipuliert DOM-Knoten
  - Einfügen, Löschen, Ändern
  - Browser 'rendert' Dokument

```
<table>
   <thead>
     <tr>
       <th>Vorname</th>
       <th>Name</th>
     </tr>
   </thead>
   <tbody>
     <tr>
       <td>Donald</td>
       <td>Duck</td>
     </tr>
   </tbody>
```

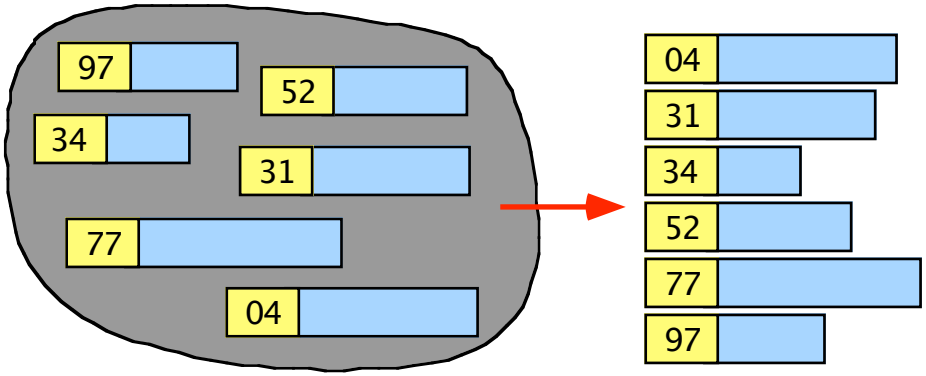


Quelle: de.wikipedia.de/wiki/Document_Object_Model

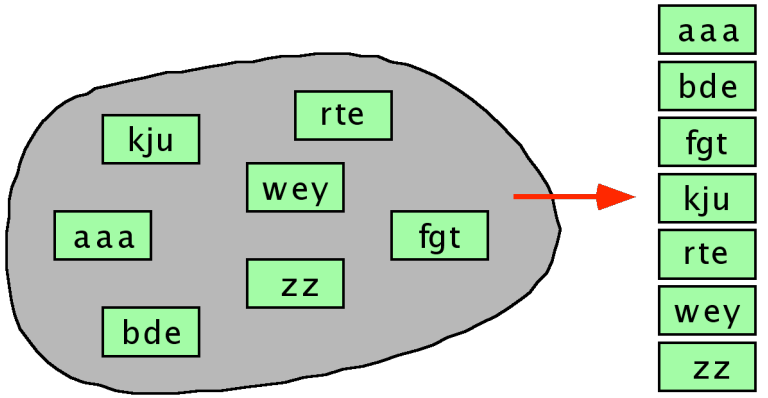
3. Algorithmische Komponenten

3.1 Sortieren

- Traditionelles EDV-Verfahren
  - algorithmische Fingerübungen
  - Abschätzungen für Rechenaufwand
  - Einfluss des Datenvolumens
- Problemstellung
  - ungeordnete Menge von Elementen,
  - evtl. nur sequentiell zugreifbar
  - Ordnung der Schlüssel gesucht

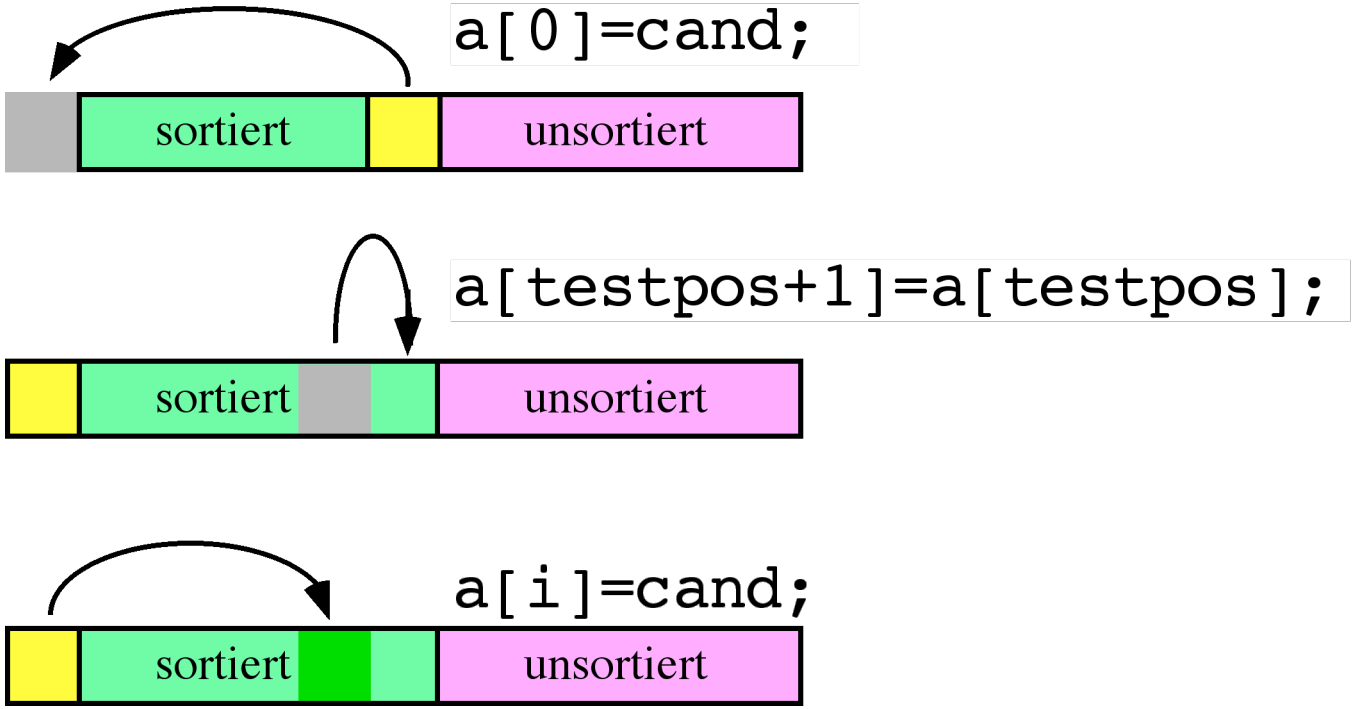


Records mit einem Schlüsselfeld



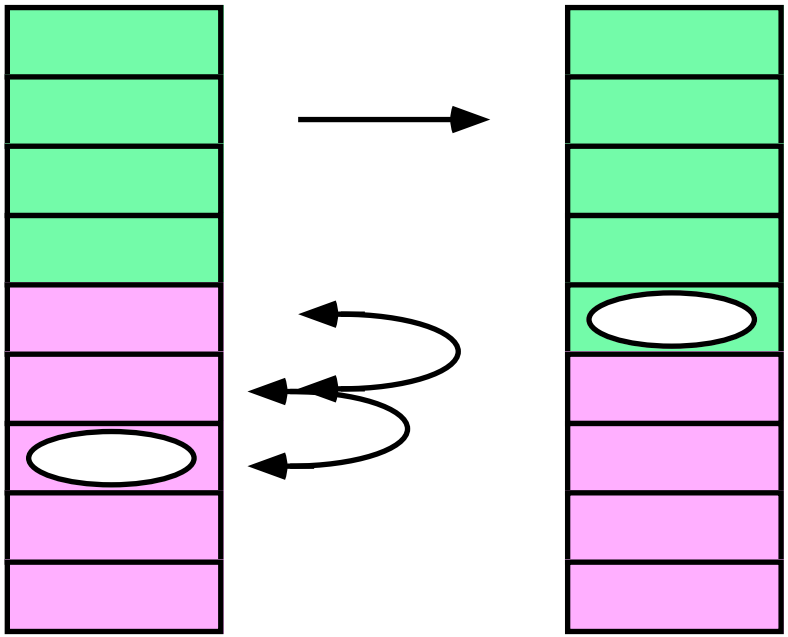
oder vollständige Elemente

- Sortieren durch Einfügen
- Zu sortieren: `item menge[N];`
  - Kandidat im linken Teil einsortieren
  - entstandene Lücke mit schon sortierten Elementen auffüllen



```
void DirektesEinfügen(item *a, int N)
{
item cand;
int candpos, testpos;
for (candpos=2, candpos<=N, candpos++)
  {
     cand= a[candpos]; a[0]= cand;
     testpos=candpos-1;
     while (cand < a[testpos])
     {
        a[testpos+1]=a[testpos];
        testpos= testpos-1;
     }
     a[testpos+1]=cand;
  }
}
```

- Sortieren durch Vertauschen ("Bubble Sort")
  - Benachbarte Elemente paarweise vertauschen, falls `a[i]<a[i+1]`
  - Analogie zu einer aufsteigenden Blase
  - bis zu der ihrem 'Gewicht' entsprechenden Höhe

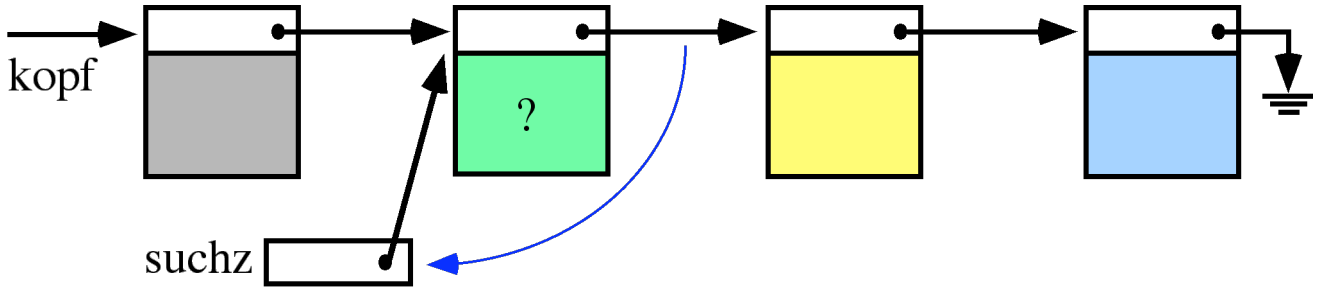


```
void bubblesort(item *a, int N)
{
item zwischen;
int fertig, cand;
for (fertig=2, fertig<N, fertig++)
{
   for (cand=N, cand>=fertig, cand--)
   {
      if (a[cand-1]>a[cand])
         { zwischen=a[cand-1];
           a[cand-1]=a[cand];
           a[cand]=zwischen;
}  }    }
```

- Verhalten der Elemente
  - leichtes Teilchen in einem Durchlauf ganz nach oben
  - schweres Teilchen sinkt mit jedem Durchlauf nur um eine Stelle
  - nur solange sortieren, bis kein Austausch mehr stattfindet
  - "Shakersort" als Variante mit abwechselnder Durchlaufrichtung
  - auch Bubblesort bewegt Elemente nur über eine kleine Distanz

3.2 Listen

- Durchsuchen einer Liste

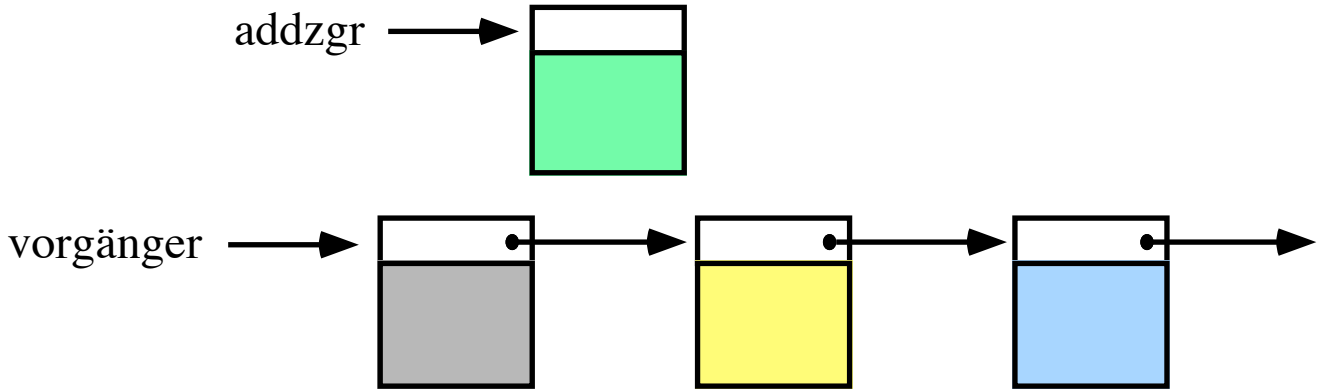



```
gefunden = NULL;
suchz = kopf;
while  (suchz != NULL)
{ if (suchz->data==gesucht)
   {
     gefunden:= suchz;
     break;
    }
   else  suchz = suchz->nxt;
}
```

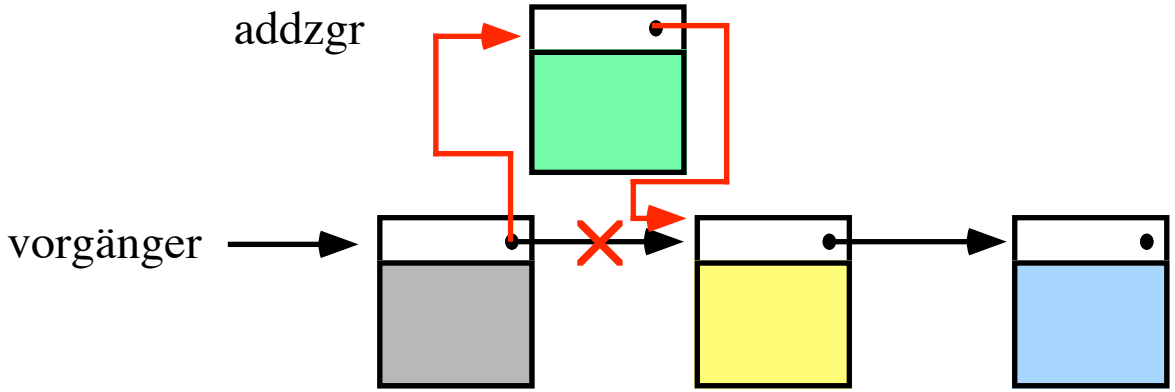
- Aufbau einer neuen Liste
  - Einlesen von `content`-Records
  - Einfügen am Kopf der Liste

```
listelem *kopf, *addzgr;
content eingabe;
...
kopf = NULL;  /* Liste leer */
while (elem_lesen(&eingabe))
{
  addzgr = elem_anlegen();
  addzgr->data = eingabe;
  addzgr->nxt = kopf;
  kopf = addzgr;
}
...
```

- Unmittelbar nach irgendeinem Vorgänger einfügen



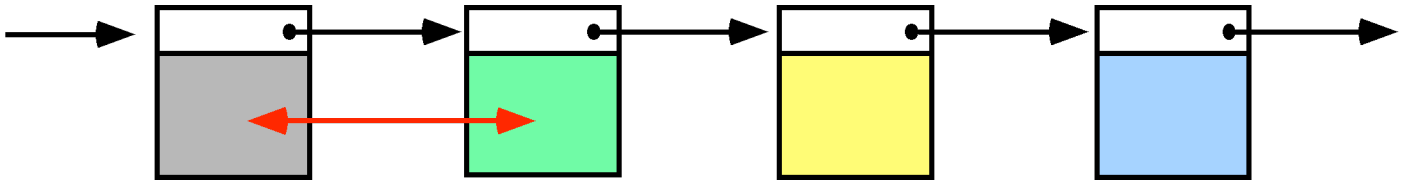

```
addzgr->nxt = vorgänger->nxt;
vorgänger->nxt  = addzgr;
```



- Zeiger auf Vorgänger wird gebraucht

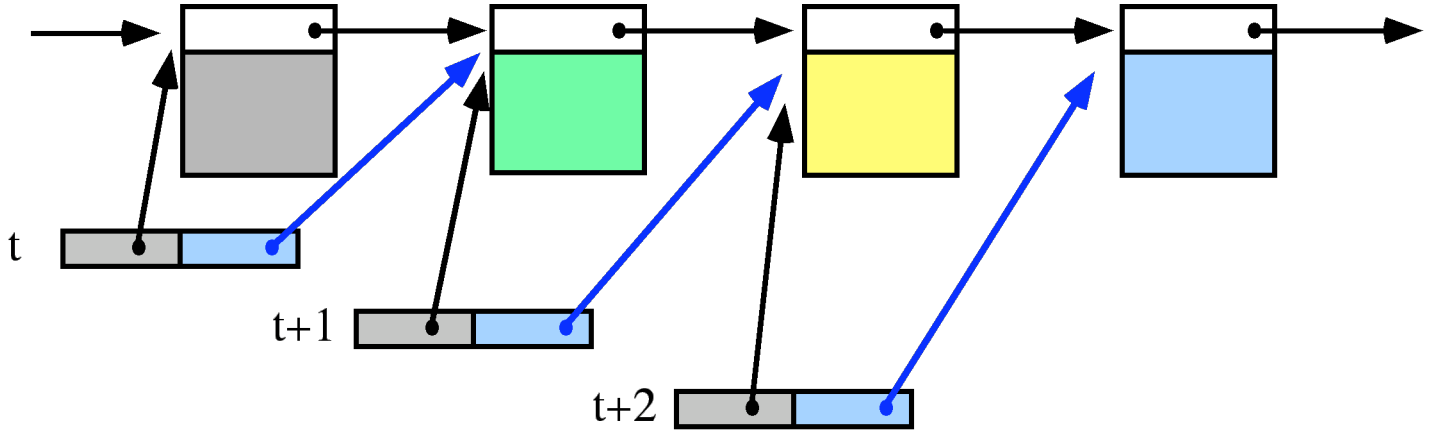
- Einfügen *nach* einem Element ist leicht
- *vor* einem Element schwierig

  => Trick: Nachher einfügen und altes Element nach vorne kopieren.

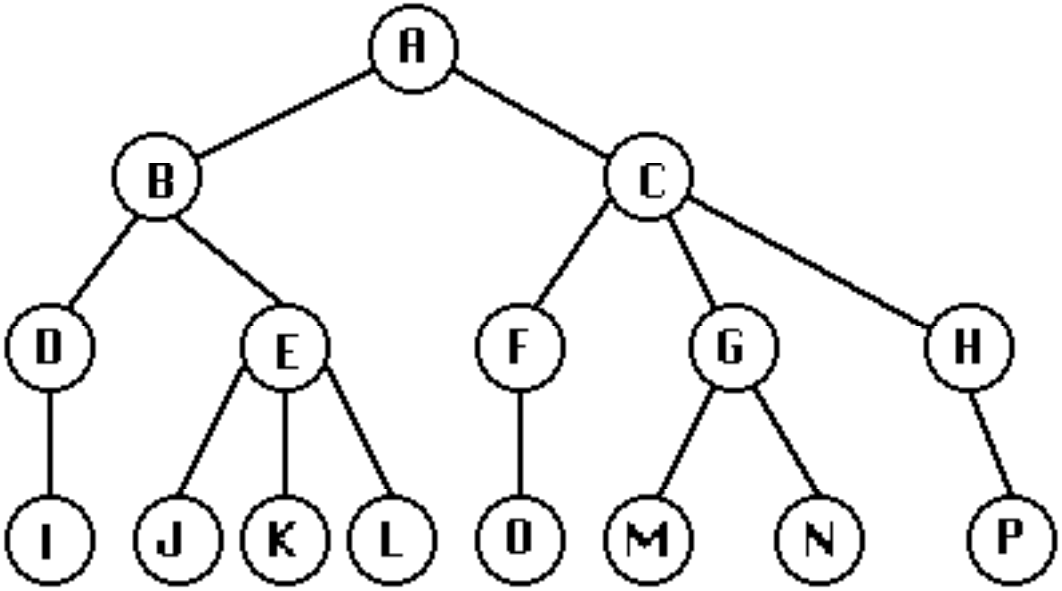


- Andere Lösung

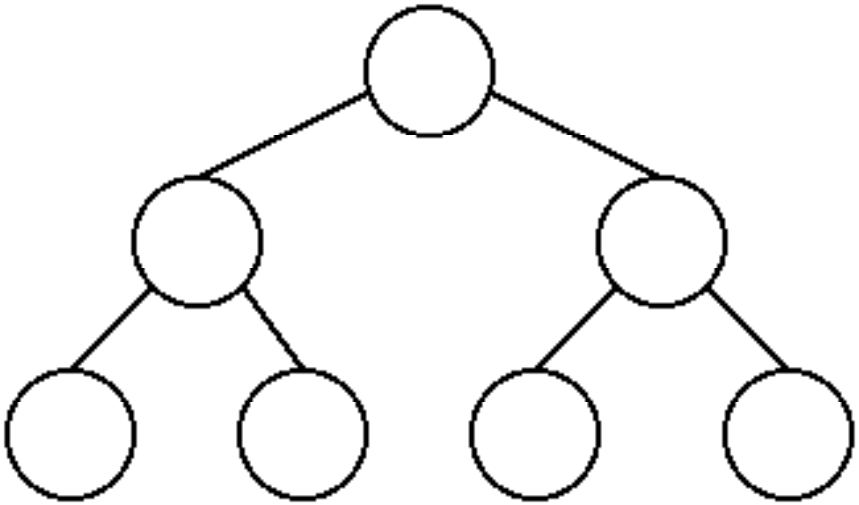
  Schleppzeiger auf Vorgänger immer mitführen



- Listen mit mehr als einem Nachfolger: Bäume



- Binärer Baum



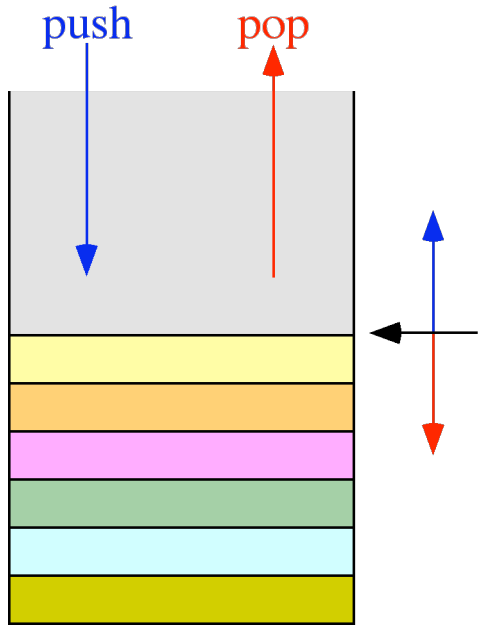
- Durchsuchen von Bäumen
  - linker Folge-Teilbaum
  - Knoten testen
  - rechter Folge-Teilbaum
  - Aufhören bei Erfolg

```
knoten *teste(knoten *mitte, elt such)
{
  knoten *help;
  if (mitte == NULL) return NULL;
  else
  { help = teste(mitte->links);
    if (help != NULL) return help;
    if (mitte->inhalt == such) return mitte;
    return teste(mitte->rechts);
} }
```

- Andere Suchreihenfolge
  - Knoten, links, rechts
  - links, rechts, Knoten

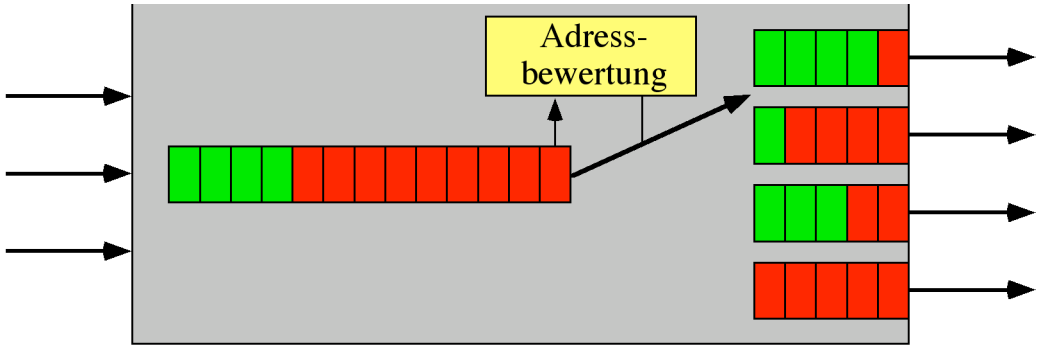
3.3 Speicherverwaltung

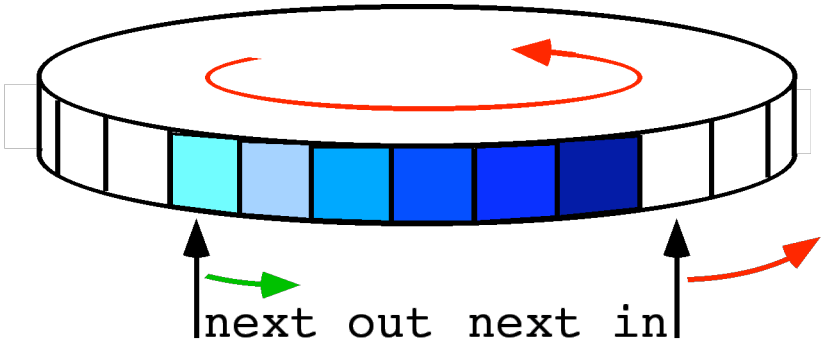
- Platz für dynamisch eintreffende Daten
- Verwaltung
  - freien Platz finden
  - gebrauchten markieren (lock)
  - nicht mehr gebrauchten freigeben
  - Fehlerroutinen
- Bearbeitungssemantik
  - Stack: last in - first out
  - Warteschlange: first in - first out
- Stack (Stapel)
  - Stack_Push legt Element in den Stack
  - Stack_Pop holt Element ab/heraus
  - Stack_Create
  - Implementierung mit Array oder mit Liste



```
#include <stdlib.h>
#include "item.h"

Item *stack;
int level, size;
Item badItem = {.text="stack_underrun"};
enum stack_error {stack_noError, stack_overflow};

void stack_Create(int maxElts)
   {stack=malloc(maxElts*sizeof(Item));
    level=0; size = maxElts;}

int stack_Push(Item theItem)
   {if (level<size)
      {stack[level++]=theItem; return stack_noError;}
    else return stack_overflow;}

Item stack_Pop()
   {if (level>0) return stack[--level];
    else return badItem;}
int main (){stack_Create(42);};
```

- Queue (Warteschlange)
  - FIFO
  - z.B. Netzwerkadapter, Router
  - in Array oder Liste
- Ringpuffer
  - 2 Zeiger: next_in, next_out
  - Schreiben inkrementiert next_in
  - Lesen inkrementiert next_out
  - inkrementieren modulo buffersize
  - am besten 8, 16, 32, 64, …

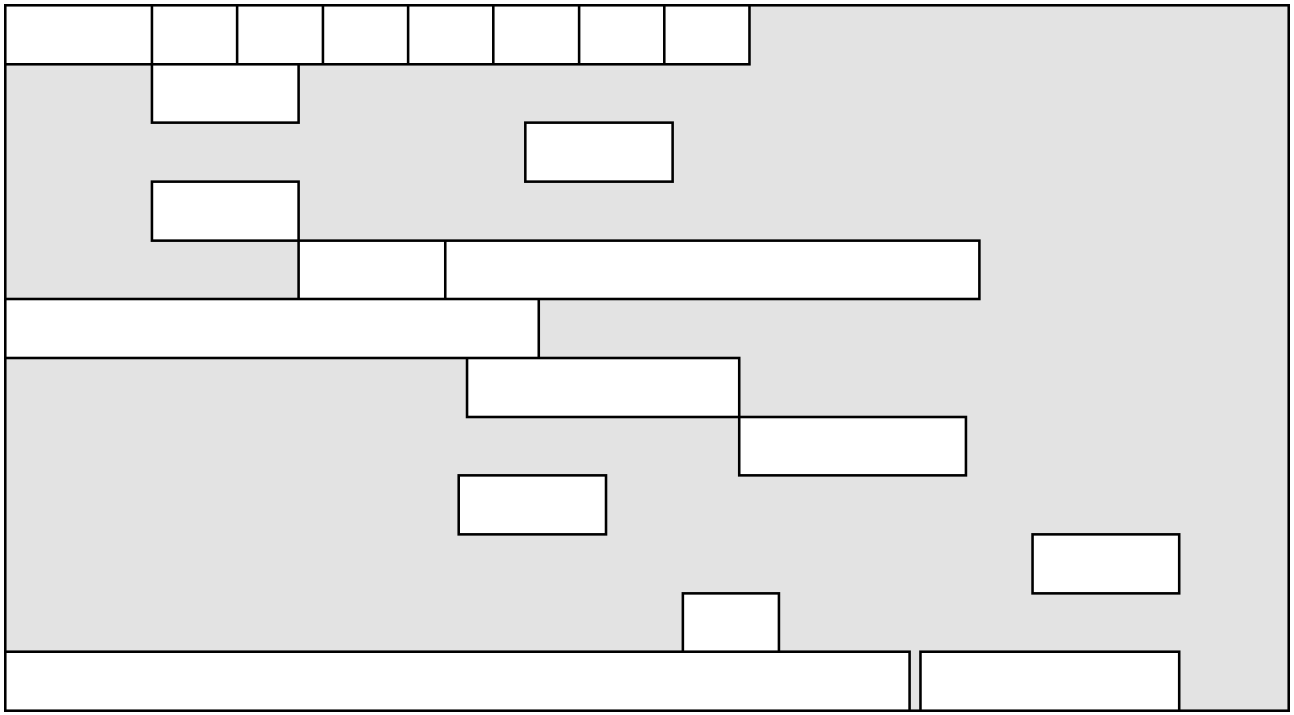





```
#include <stdlib.h>      /* nach Sedgewick */
#include "Item.h"

typedef struct QUEUEnode* link;
struct QUEUEnode { Item item; link next; };
link head, tail;

void QUEUEinit() { head = NULL; }
int QUEUEempty() { return head == NULL; }
```

```
link NEW(Item item, link next)
  { link x = malloc(sizeof *x);
    x->item = item; x->next = next;
    return x;
  }

enQueue(Item item)
  { if (head == NULL)
      { head = (tail = NEW(item, head)); return; }
    tail->next = NEW(item, tail->next);
    tail = tail->next;
  }
Item deQueue()
  { Item item = head->item;
    link t = head->next;
    free(head); head = t;
    return item;
  }
```

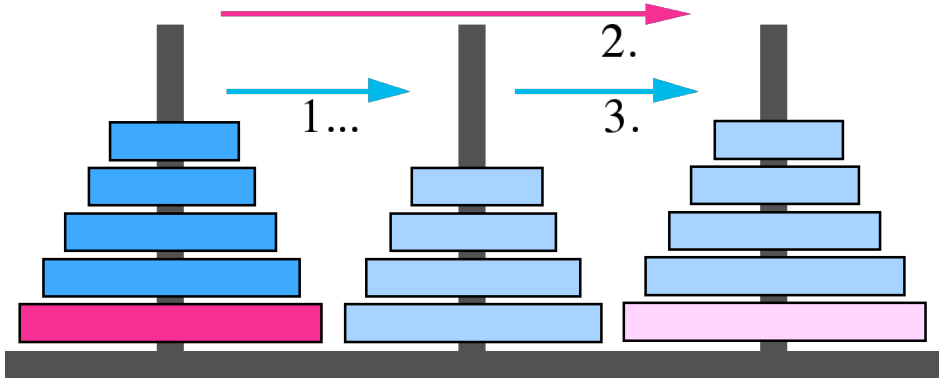
- Heap (Halde)
  - beliebige Speicherbereiche
  - anlegen mit `void *malloc(size_t size)` aus stdlib.h
  - freigeben mit `free(void *pointer)`



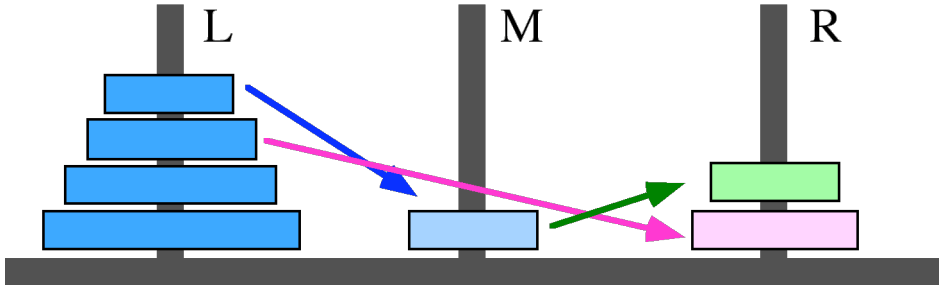
- Implementierung
  - großer Bereich des Haupspeichers
  - Liste mit belegten Blöcken
  - Freispeicherliste
  - Fragmentierung sorgfältig managen

3.4 Rekursion

- Funktion wird in sich wieder aufgerufen
  - direkt oder indirekt
- Problem auf einfacheres zurückführen
  - wiederholen bis zum trivialen Problem
  - in jedem Schritt etwas leichtes machen
- wo haben wir das schon benutzt?
- Beispiel: Türme von Hanoi
- Regeln:
  - Scheibenturm von links nach rechts bringen
  - immer nur eine Scheibe bewegen
  - nur kleinere Scheiben auf größere legen



- Lösungsansatz:
  - Bewegung des Turmes mit Höhe N auf Turm mit Höhe N-1 zurückführen,
  - Turm(N-1) auf mittleren Stock bringen
  - verbleibende Scheibe nach rechts
  - Turm(N-1) nach rechts
- Rekursive Strategie:
  - triviale Lösung für Turm(0)
  - Lösung zu Turm(1) = Lösung zu Turm(0) + eine Scheibe bewegen
  - Lösung zu Turm(N) = Lösung zu Turm(N-1) + Scheibe N bewegen



- Züge für eine Turmhöhe von 4

L->M, L->R, M->R, L->M, R->L, R->M, L->M, L->R,
M->R, M->L, R->L, M->R, L->M, L->R, M->R,

```
#include <stdio.h>

typedef char Stock;  /* 3 Stöcke: 'L','M','R' */

void BewegeScheibe(Stock von,Stock nach)
{ Printf("%c",von); Printf("->%c",nach);
  Printf("%c",',');
}

void  BewegeTurm(int restHoehe, Stock von,
                 Stock zwisch,Stock nach)
{ if (restHoehe<=0) return;
  BewegeTurm(restHoehe-1,von,nach,zwisch);
  BewegeScheibe(von, nach);
  BewegeTurm(restHoehe-1,zwisch,von,nach);
  printf("\n");
}

main()
{  BewegeTurm(4,'L','M','R'); }
```

3.5 Objekte, Patterns und Frameworks

3.5.1 Grundzüge des objektorientierten Programmierens

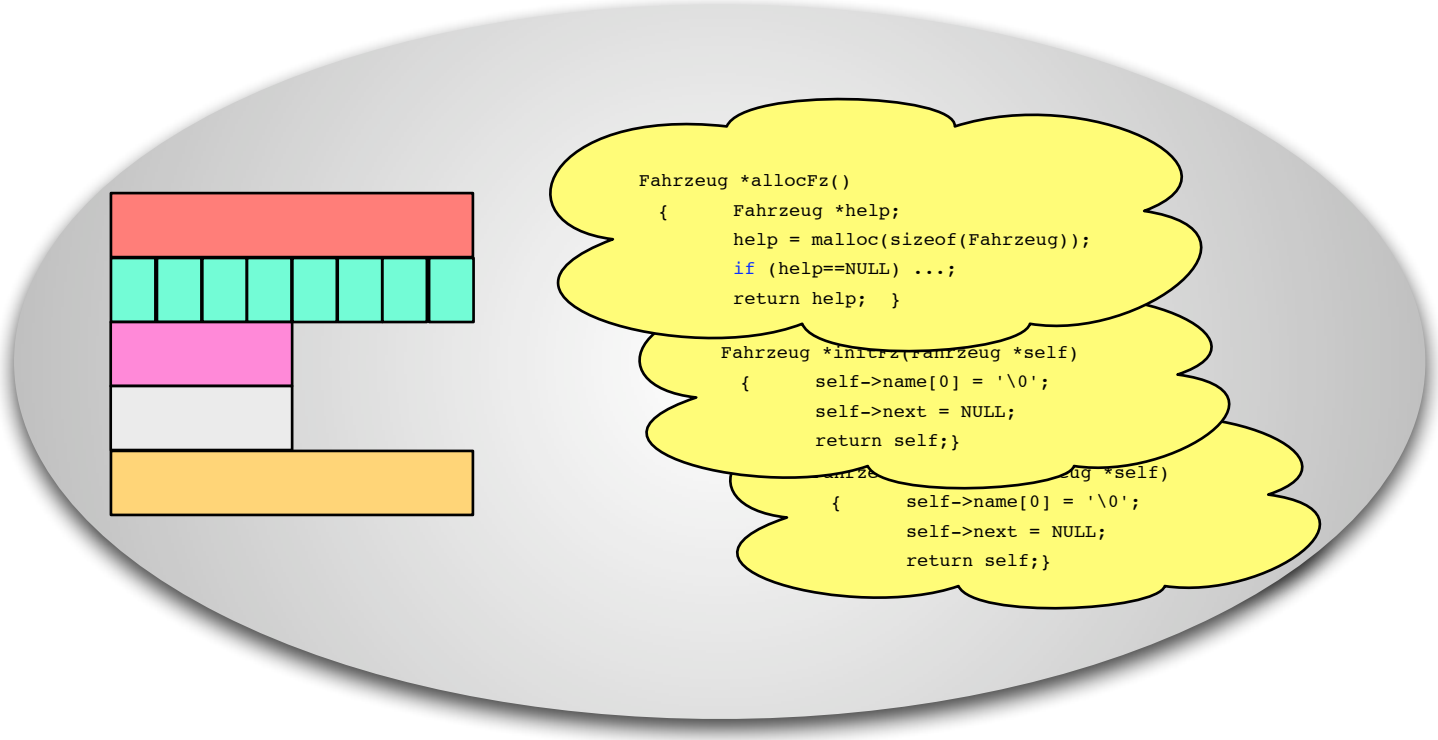
- Datenstrukturen
  - Inhalt als Attribute
  - Funktionen zur Inhaltsmanipulation
  - Semantik steckt in den Funktionen

```
typedef struct fzstruct
{  char name[32];
   enum {Lok,Perswg,Gueterwg} art;
   int Baujahr;
   struct Fahrzeug *next;
} Fahrzeug;
```

```
Fahrzeug *lok;
…
lok = initFz(allocFz());
lok->next = initFz(allocFz());
```

```
Fahrzeug *allocFz()
{  Fahrzeug *help;
   help = malloc(sizeof(Fahrzeug));
   if (help==NULL) ...;
   return help;
}

Fahrzeug *initFz(Fahrzeug *self)
{  self->name[0] = '\0';
   self->next = NULL;
   return self;
}
```

- Objekte: Struktur+Funktionen
  - als eigenständige Einheit des Programmierens
  - Klassiker: Initialisierung einer Struktur, Vergleich, …
  - datenstruktur-orientierte Modularisierung



- Beispiel Objective-C
  - Brad Cox, Tom Love, 1980 …
  - inspiriert von Smalltalk
  - NeXTStep, OpenStep, GNUStep, Cocoa, iPhone, iOS

- Klasse = Strukturdeklaration + Funktionen
  - Funktionen zum:
    - Initialisieren, Löschen, …
    - evtl. Felder zugreifen
    - Listenoperationen
  - heißen Methoden, Selektoren, Messages

  => Konzept

```
#import <Cocoa/Cocoa.h>
@interface fahrzeug : NSObject {
   char name[32];
   enum {Lok, Perswg, Gueterwg} Art;
   int Baujahr, Gewicht;
   fahrzeug *next; }
- (fahrzeug *)init;
- (void) add: (fahrzeug *)fz;
- (void) setweight:(int)wght;
- (int) getweight;
- (int) cmpwght;
- (fahrzeug *)getnext;
@end
```

```
#import "fahrzeug.h"
@implementation fahrzeug
- (fahrzeug *)init
{ [super init];
   name[0] = '\0';
   ...
}
- (void)add:(fahrzeug *)fz
   {next = fz; }
- (void) setweight:(int)wght
{  Gewicht = wght;}
- (int) getweight
{  return Gewicht}

- (fahrzeug *)getnext
{  return next; }

- (int) cmpwght
{ if (self == nil) return 0;
  else return Gewicht +
        [next cmpwght];       }
@end
```

- Instanzen / Instantiierung
  - dynamisches Anlegen: alloc
  - meist Initialisieren: init
  - Referenzen in Zeigern halten

```
fahrzeug *zug, *z1;

zug = [[fahrzeug alloc]init];
z1  = [[fahrzeug alloc]init];
[zug add:z1];
```

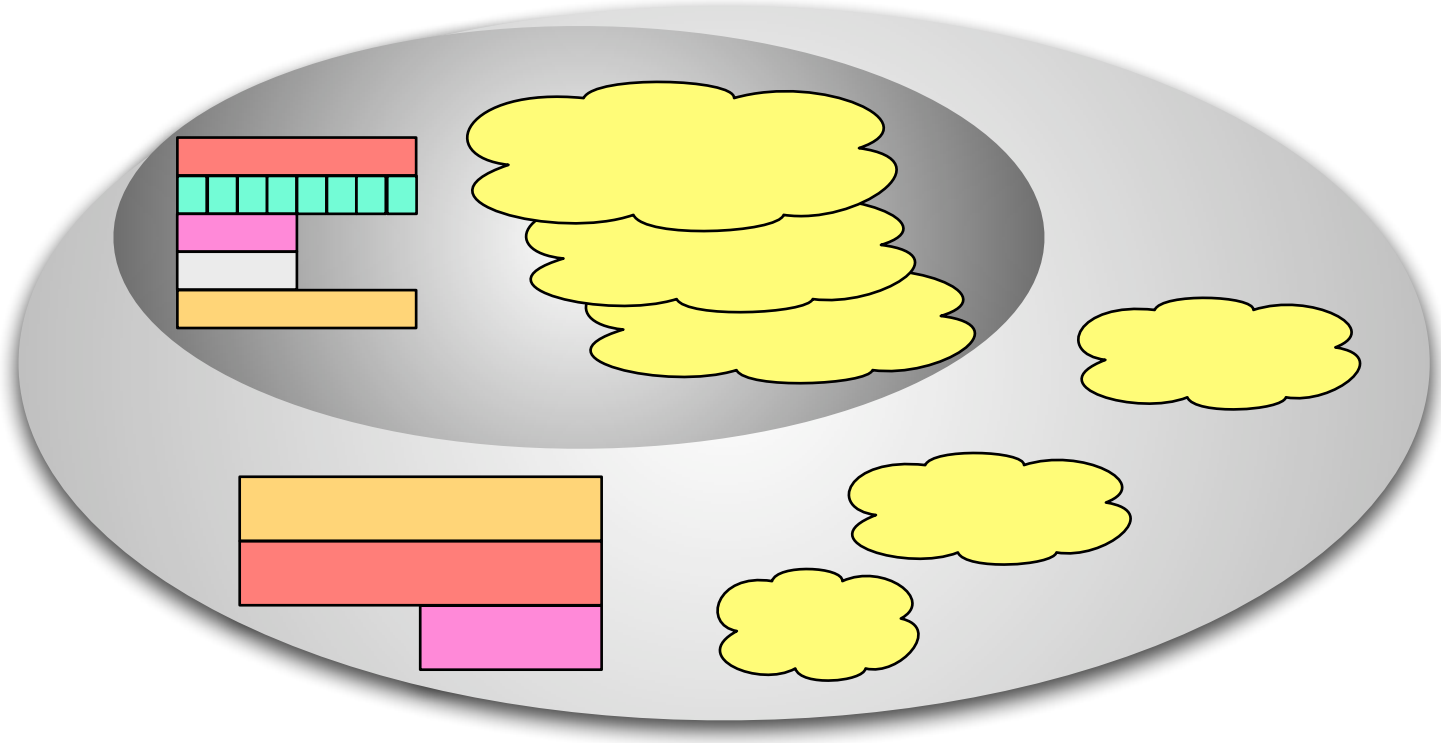
- Nachrichten senden
  - entspricht Funktionsaufruf
    `[<obj> <message>]`
  - mit Parametern:
    `[<obj> <message>:<parm>]`
    `[<obj> <message>:<parm> p2:<parm> p3: ...]`
  - Namenskonvention für Parameter 2 ... n: `andp2` ...
  - Abkürzung: [<obj> <message>:<parm>:<parm>: ...]

```
- (fahrzeug *)init
{ [super init];
  name[0] = '\0';
  next = nil;
  Gewicht = 0;
  return self;     }
```

- Accessor-Methoden
  - in den Methoden der Klasse mit Feldname
  - klassisch: `<instance>.<property>`
  - Accessor `[<object] <feldzugriff>]`
  - Namenskonvention für Methode `get<feld>` bzw. `set<feld>`

```
- (void)setweight:(int)wght;
- (int) getweight;
```

- Vererbung
  - Klassen erweitern
  - Basisklasse + neue Felder und Methoden



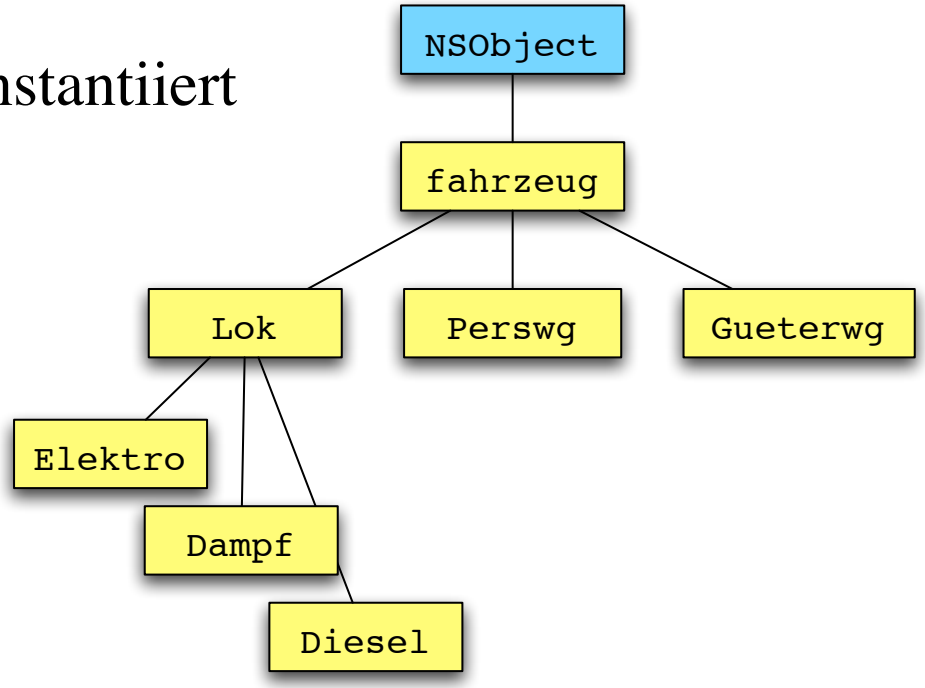
  - Felder und Methoden der Basisklasse weiter verwendbar
  - Begriff: Superklasse
  - self und super

- Methoden
  - erben
  - neue hinzufügen
  - manche überschreiben
  - Scope: Blatt zur Wurzel

```
#import <Cocoa/Cocoa.h>
#import "fahrzeug.h";

@interface lok : fahrzeug
{
   enum {Elektro, Diesel, Dampf} prinzip;
   int leistung;
   int last;
}
- (lok *) init: (int) lst andlast: (int) load;
- (int) add: (fahrzeug *) fz;

@end
```

```
#import "lok.h"

@implementation lok
- (lok *)init:(int)lst andlast:(int)load
{
   [super init];
   prinzip = Elektro;
   leistung = lst;
   last = load;
   return self;
}

- (int) add:(fahrzeug *) fz
{
  if (last<[fz cmpwght]) return 0;
   else
   { [super add:fz];
     return 1;
   }
}
@end
```

- Klassenbaum
  - Alle Klassen erben von einer Wurzelklasse, hier: NSObject
  - möglichst viele Methoden erben
  - abstrakte Klassen werden nicht *selbst* instantiiert
- Klassenmethoden
  - zum allgemeinen Klassenmanagement
  - z.B. Konstruktor (`alloc`)
  - `+ (<type>) <name>: ...`
- Polymorphismus?
  - von mehreren Klassen erben
  - nicht wirklich notwendig
  - nicht in allen OO-Programmiersprachen
  - in manchen Hilfskonstrukte



3.5.2 Frameworks am Beispiel

- Funktionen und Strukturen in Libraries
  - häufig verwendet
  - Listen, Dateizugriff, Ein/Ausgabe, Fenster, …
  - auch API genannt
  - kann zur Abstraktion verwendet werden
  - Portabilität
- Framework
  - Menge von Klassen
  - Wiederverwendbarkeit
  - Erweiterbarkeit: überschreiben, spezialisieren, …
  - gibt Programmstruktur vor
  - "Klasse Programm"
- Kontrollumkehr
  - Programm wird zur Funktionssammlung
  - Funktionen im Programm werden gerufen
  - Programmierer *füllt* vorgegeben Funktionen

- Frameworks in iOS und MacOS
  - Core Image, OpenGL, Quartz, QuickTime, …
  - AppKit: Windows, Buttons, Menus, …
- Beispiel Foundation.h
  - aus NeXTStep: Namen NS*
- Numbers, Strings, Collections
  - NSData
  - NSString: suchen, vergleichen, konkatenieren
  - NSArray, NSDictionary, NSSet
  - NSCharacterSet, NSScaner (Zahlen aus Strings extrahieren)
- Operating System Services
  - NSFileManager, MSPathUtilities
  - NSThread, NSProcessInfo
- Memory Management für Objekte
  - NSAutoReleasePool
- URLs

- Beispiel NSString
  - auch für Unicode Char, UTF8 und UTF16
  - unveränderlich: NSString
  - mutable subclass: NSMutableString

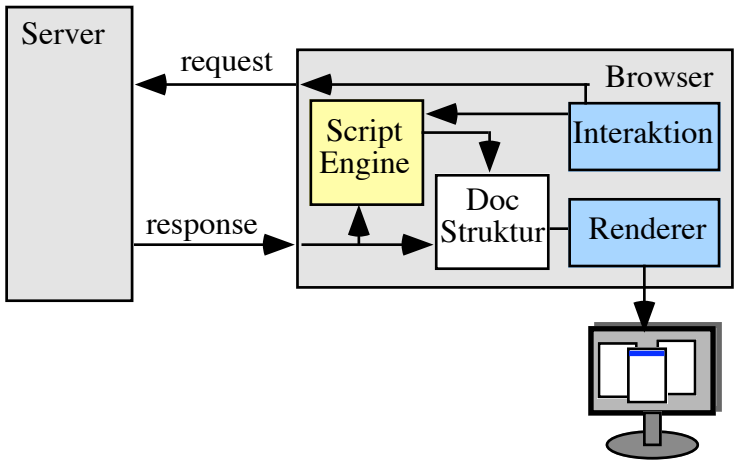
```
NSString *hString = @"Hello";
NSString *hwString = [hString stringByAppendingString:@", world!"];
```

- Methoden
  - Erzeugen (auch aus C-Strings, URLs oder Dateien)
  - Länge
  - Teilstrings suchen und herausholen
  - Teilen und Konkatenieren
  - Vergleichen
  - Pfade für Dateisystem

```
NSString *s10 = @"blue10";
NSString *s2 = @"blue2";
NSComparisonResult res;

res = [s10 compare:s2];
// res = -1 (NSOrderedAscending)

res = [s10 compare:s2 options:NSNumericSearch];
// res = 1 (NSOrderedDescending)
```

3.5.3 Patterns

- Gang of four: Design Patterns: Elements of Reusable … [1994]
- Lösungen für typische Problemstrukturen
  - generalisiert, wiederverwendbar
  - 'best practice'
  - "über" Datenstrukturen und Algorithmen
  - auch zur Prozesskommunikation
- Ein berühmtes architectural pattern: MVC
- Model
  - Verhalten und Daten
  - application domain
- View
  - Darstellung des Modells zur Interaktion
  - Display, Rendern
- Controller: reagiert auf Events
  - ändert Modell
  - auch nachrichtenbasiert



- Creational Patterns
  - Objekte erzeugen
  - Factory, Builder, Object Pool, Prototype
- Singleton
  - Menge mit genau einem Element
  - hier: nur eine Instanz einer Klasse
  - einziger Zugangspunkt
  - z.B. beim parallelen Programmieren
  - kapselt gemeinsam genutzte Resource

```
static Singleton *sharedSingleton = nil;

+ (Singleton*)sharedManager
{
    if (sharedSingleton == nil) {
        sharedSingleton = [[super alloc] init];
    }
    return sharedSingleton;
}
// Methoden zur Arbeit mit der Klasse folgen …
```

- Structural Patterns
  - Beziehungen zwischen Elementen
  - Bridge, Adapter
  - Composite und Aggregate
  - Proxy, Facade, Extensibility (Framework pattern)
- Behavioural Patterns
  - Chain of responsibility: Events erhalten und weitergeben
  - Observer (publish/subscribe): für Events registrieren
  - Iterator greift auf Felder nacheinander zu
- Concurrency Patterns
  - konzeptuell parallele Ausführung (threads)
  - Active Object, Monitor, Leader Follower, …
  - Reactor
  - Scheduler
  - Thread Pool