TECHNISCHE UNIVERSITÄT DRESDEN

Fakultät Informatik | Institut Systemarchitektur | Professur Rechnernetze

WS 2013
LV Informatik-I für Verkehrsingenieure

5. Programmierungstechnik
5.1 Übersicht

Dr. rer.nat. D. Gütter

Mail: Dietbert.Guetter@tu-dresden.de
WWW: wwwpub.zih.tu-dresden.de/~guetter/

Hinweis

Informatik-I

- Übersicht zur Programmierungstechnik, kein Programmierkurs
- Algorithmierung – Grundstrukturen (s. Kap. 3.1)
- Algorithmierung – Beispiele (s. Kap. 3.2)
- Übersicht zur Softwaretechnologie (s. Kap. 4)

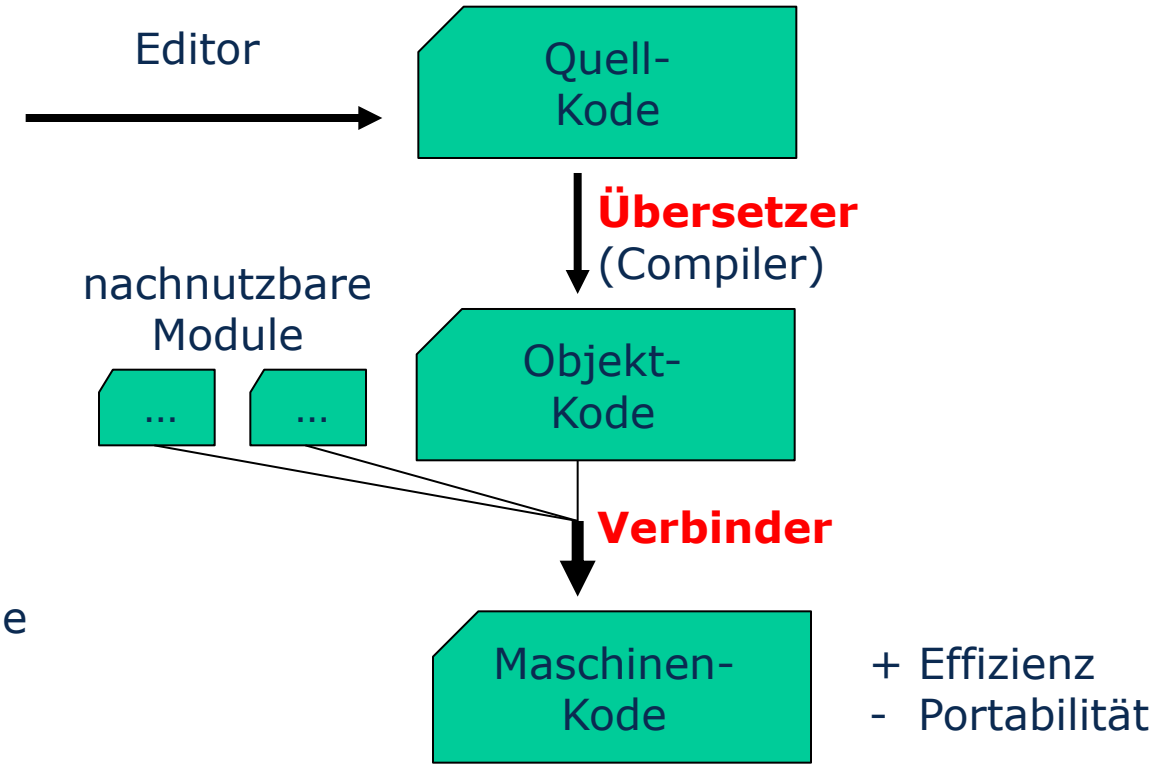
Informatik-II

- Softwaretechnologie - Vertiefung
- Programmierungstechnik – Vertiefung (Java im Detail und zugehöriges Praktikum)

Programmierung (1)

Programmierer

- löst Problem
- formuliert Quellprogramm in einer Programmiersprache



Programmierung (2)

Programmierer

- löst Problem
- formuliert Quellprogramm in einer Programmiersprache

Editor → Quell-Kode

Starten **Interpreter**

Zyklus
- Lesen Quellprogramm-"Zeile"
- Inhalt - Interpretation
- Inhalt – Ausführung
- Bestimmung der Nachfolge-"Zeile" ggf. Ende Programmlauf

\+ Portabilität
\- Effizienz

Programmierung (3)

Programmierer muss Problemlösungskompetenz besitzen,

d.h. er muss das Problem erkennen
und schrittweise in Teilprobleme untersetzen,

und diese mittels der (einfachen) Elemente
von Programmiersprachen lösen.

aber er

- entwickelt (normalerweise) keine Programmiersprachen.
- muss Programmiersprache(n) nur 1x lernen.

z.B. *„Text ausgeben"*	in	Pascal	writeln(´Hallo Welt!);
	in	*C*	printf("Hallo Welt!\n");
	in	Javascript	document.write("Hallo Welt!");

- muss nicht wissen, wie ein Compiler/Interpreter arbeitet.

Programmiersprachen (1)

Programmier-sprachniveau	Objekte	Nutzung
Maschinen-programmierung (s. Kap. 2 und 3)	Befehle, Adressen, Daten (Binärnotation)	Zugriff auf alle Ressourcen eines Computertyps; zu mühsam, veraltet
Assembler-programmierung, z.B. **MASM**	Befehle, Adressen, Daten (symbolische Notation) „Assembler" übersetzt in Maschinenkode	Zugriff auf alle Ressourcen eines Computertyps; mühsam, nur für spezielle Betriebssystemroutinen
Problemorientierte Sprachen, z.B. **C**, **C++**, **C#**, …	Anweisungen, vielfältige Datentypen „Interpreter" und „Compiler" (Übersetzer)	Transparente Hardware (Rechner- und Betriebssystem-unabhängigkeit); Hardwarezugriff eingeschränkt universelle Problembehandlung
Spezialsprachen, z.B. **GPSS**	sprachspezifisch; bei GPSS Warteschlangen, Forderungsströme, …	Effiziente Lösung spezieller Problemfelder, bei GPSS: Simulation von Netzwerken der Bedienungstheorie

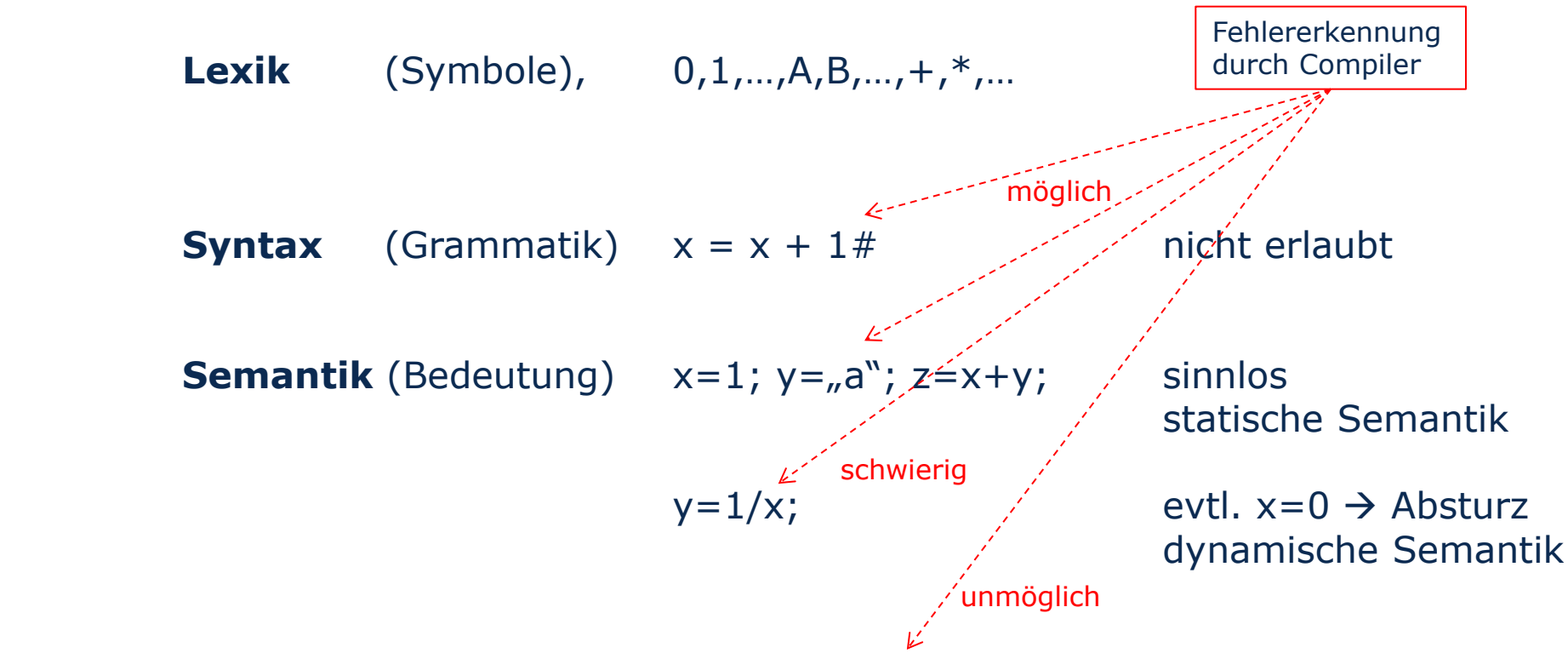
Programmiersprachen (2)

Programmier-sprachniveau	Objekte	Nutzung
Stapel-programmierung, z.B. **Bourne-Shell** (Unix)	Kommandos, Prozesse, Dateien (Kommandointerpreter)	Automatisierung/Organisation der Abarbeitung von Prozessen auf einem Computer
…	…	…
Orchestrierung von Netzwerkdiensten, z.B. **WS-BPEL**	Dienste im Netzwerk; bei WS-BPEL Kombination von Webservices	komplexe Workflowprogrammierung „Internet der Dienste“ z.B. weltweite Dienstverknüpfung von Reisebüros, Hotels, Airlines, Auto-Verleih, …
…	…	…

Schwerpunkt bei Lehrveranstaltung Informatik für Verkehrsingenieure

- **Problemorientierte Sprachen**

Programmiersprachen – Kennzeichen/Beispiele



Lexik (Symbole), 0,1,…,A,B,…,+,*,…

Syntax (Grammatik) x = x + 1# — nicht erlaubt

Semantik (Bedeutung) x=1; y=„a"; z=x+y; — sinnlos, statische Semantik

y=1/x; — evtl. x=0 → Absturz, dynamische Semantik

Logik Brutto = Netto – Mehrwertsteuer;

Pragmatik (Nutzbarkeit) z.B. Java-Lösungen sind plattformunabhängig, haben aber Leistungsnachteile.

Programmierstil "Deklarative Programmierung"

Funktionale Programmierung

- Programme werden als vernetzte Funktionen realisiert.
- Wichtiges Mittel ist die Rekursion

z.B. „Berechnung der Fakultät n! natürlicher Zahlen n >1"

```
function fak(n)
if n > 1  then output: n*fak(n-1) else  output: 1
```

Logische Programmierung

- Programm besteht aus logischen Regeln (wenn…, dann …)
- wichtig für Probleme der künstlichen Intelligenz

! Deklarative Programmierung wird in dieser LV nicht behandelt !

Programmierstil "Imperative Programmierung"

Programm

- besteht aus einer Menge von Anweisungen (Befehlen) und Regeln für die Reihenfolge der Ausführung
- zustandsorientierte Programmierung
  - Zustand entspricht Wertbelegung aller Programmvariablen zu einem konkreten Zeitpunkt
  - taktweise Befehlsabarbeitung → jeweils neue Zustände
- Programme werden i.a. unterteilt in
  - Hauptprogramm und
  - mehrfach verwendbare Prozeduren (Unterprogramme), welche ihrerseits Prozeduren nutzen können
- Programme können selbst mehrfach nutzbar sein. → Module in großen Softwareprojekten

Elemente eines imperativen Programms (1)

Speicherreservierungen (für Daten)

Daten werden im Programm genutzt

- über Angabe eines Bezeichners z.B. x
  oder über Angabe eines Direktwertes z.B. 1

Die Daten benötigen Hauptspeicherplatz.
Im Programm gibt es deshalb Anweisungen zur Speicherreservierung.

- z.B. ab (symbolischer) Adresse x int x;
  2 Byte Speicherplatz reservieren
  für ganze Zahl

- Speicherreservierung für Direktwerte erfolgt transparent durch Compiler bzw. Interpreter.

Symbolische Adressen gibt es nur in Quellprogrammen;
übersetzte Programme arbeiten mit realen Adressen.

Elemente eines imperativen Programms (2)

Ergibtanweisungen (Wertzuweisungen)

notiert wie mathematische Gleichungen, aber andere Bedeutung

1. Lesen der Werte der Variablen auf der rechten Seite
2. Ergebnisberechnung für rechte Seite
3. Ergebnis auf Platz der links benannten Variable speichern

z.B.

```
x=x+y;  // Summe der Werte von x und y
        // ergeben den neuen Wert der Variable x

x=1;    // Variable x erhält den (Direkt-)Wert 1
```

In übersetzten Programmen entspricht eine Ergibtanweisung vielen Maschinenbefehlen.
→ effiziente und übersichtliche Programmierung

Elemente eines imperativen Programms (3)

Steueranweisungen

regeln den Programmfluss
(Reihenfolge der Ausführung der Anweisungen).

Unterprogramm-Aufrufe

Unterprogramme fassen häufig benutzte Anweisungfolgen zusammen und können als Funktion bzw,. Prozedur ähnlich wie Anweisungen genutzt werden.

Compiler/Interpreter bieten vorgefertigte Unterprogramme für Routineprobleme an, z.B. für Dateiarbeit und Bildschirmnutzung.

Programmierer können aber auch eigene Unterprogramme schreiben.

Kommentare

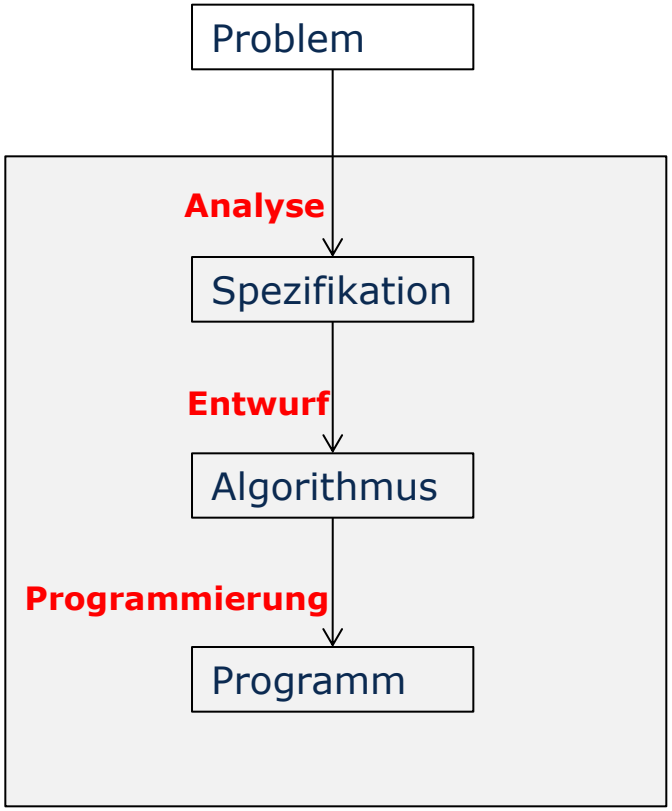
zur besseren Lesbarkeit für Programmierer
(für Übersetzungsvorgang ohne Bedeutung)

Programmiersprachen - Auswahlkriterien

- Sprache muss ausreichend leistungsfähig sein.
  - Eignung für konkretes Anwendungsproblem
  - Übersichtlichkeit
  - Effizienz
- Gute Entwicklungsumgebung für das Zielsystem muss vorliegen.
  - Editor
  - Compiler/Interpreter
  - Testwerkzeuge
- Der eigene Einarbeitungsaufwand muss möglichst gering sein.
- Evtl.
  müssen Vorgaben durch Auftraggeber berücksichtigt werden.

Programmentwicklung

- Mehrere Etappen
- Notation muss jeweils eindeutig und vollständig sein!



Was ist gegeben? (Vorbedingungen)
Was soll erreicht werden? (Nachbedingungen)

Wie soll das Ziel erreicht werden?

Algorithmus,
in rechnerverarbeitbarer Form

Programmspezifikation

Notation der Rahmenbedingungen eines Programms

1. Formulierung von **Vorbedingungen**
   für die Ausführung eines Programmes
   - Eingabeparameter, Reihenfolge, ggf. Zeitschranken, …
   - z.B.
     Es liegen n relle Zahlen in ungeordneter Reihenfolge vor.
2. Formulierung der **Nachbedingungen**
   - Ausgabeparameter, Reihenfolge, ggf. Zeitschranken, …
   - z.B.
     Die n Zahlen sollen größengeordnet ausgegeben werden.

Nicht in der Spezifikation enthalten sind Informationen zur Problemlösung.
→ **Algorithmus**

Algorithmus

Definition

Ein Algorithmus ist

ein präzise und eindeutig formuliertes Schema, das

unter Verwendung von endlich vielen ausführbaren … Arbeitsschritten … zur Lösung einer Klasse gleichartiger Probleme

auch von einer Maschine schrittweise abgearbeitet werden kann.

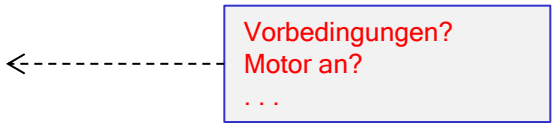
Den Vorgang der Abarbeitung bezeichnet man als Prozess,
die ausführende Maschine als Prozessor.

W. Schmitt in

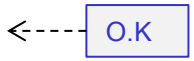
Lehr- und Übungsbuch Informatik
Fachbuchverlag Leipzig im Carl Hanser Verlag, 2001
Herausgeber: C.Horn, I.Kerner, P.Forbrig

Algorithmus – Diskussionsbeispiel

Tempomat (für Automobile)



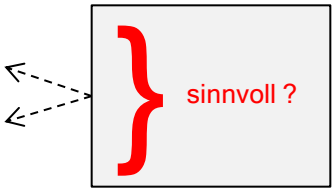
Einstellen Sollgeschwindigkeit durch Fahrer
(Eingabe aus Algorithmus-Sicht)



Danach periodisch

1. Ermitteln Fahrgeschwindigkeit (Tachometer)
   (Eingabe aus Algorithmus-Sicht)
2. falls Fahrgeschwindigkeit niedriger als Sollgeschwindigkeit
   → Vollgas
   (Ausgabe aus Algorithmus-Sicht)

   sonst
   → Gas wegnehmen
   (Ausgabe aus Algorithmus-Sicht)

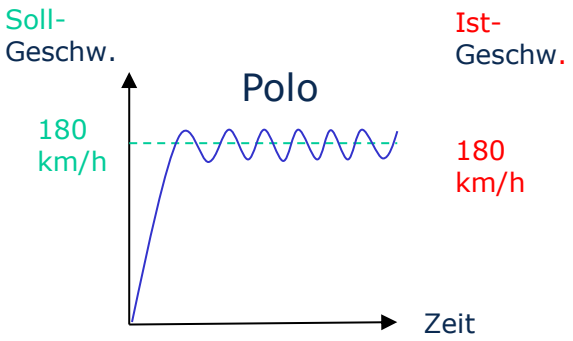


Algorithmus – Zielstellung erfüllt ?

Ja,

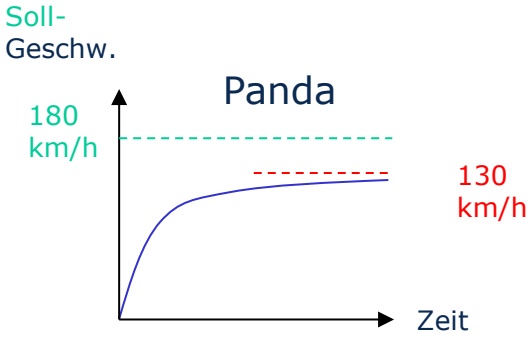
nur für Autos

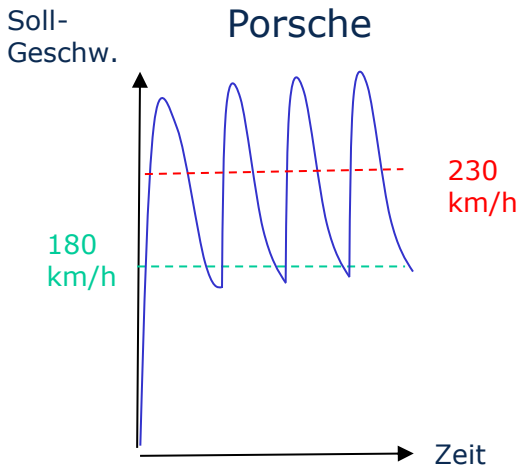
mit optimaler Motorleistung



Modell stark vereinfacht

Nein, bei sonstigen Motorisierungen





Validierung

Prüfung, ob Algorithmus-Ausführung das gegebene Problem löst,
i.a. nicht formal lösbar

z.B.

Analyse unkorrekt → fehlerhafte Spezifikation,
Algorithmus erfüllt Spezifikation,
löst aber das Problem nicht.

Validierung erfordert

- Theorie zur Anwendungsproblematik (Modellierung)
- Kompetenz des Anwenders
- Spezifische Problemanalyse
- Evtl. Testarbeiten
- Evtl. Simulationen, falls Testarbeiten erschwert sind
  (z.B. Steuerung einer neuen Raketengeneration am Boden)

→ **Validierung ist i.a. kein Gegenstand der Informatik**
und wird deshalb in dieser Lehrveranstaltung nicht weiter betrachtet.

Verifikation (Nachweis der Fehlerfreiheit)

Algorithmus-Verifikation

- Werden die Forderungen der Spezifikation durch den gewählten Algorithmus erfüllt?
  - Evtl. Entwurfsfehler im Algorithmus
  - Feststellung der Korrektheit
    i.a. formal realisierbar (aufwendig, aber 100%-ig sicher)

Programm-Verifikation

- Werden die Forderungen der Spezifikation bei der Programmausführung erfüllt?
  - Evtl. Programmierfehler,
    evtl. Nutzung fehlerbehafteter Betriebssystemrufe, …
  - Feststellung der Korrektheit durch qualifizierte Testreihen

Nicht in allen Fällen ist eine 100%-ige Verifikation möglich!

Algorithmen - Grundelemente

Arbeitsschritte

- Ein- und Ausgabeanweisungen
- Verarbeitungsanweisungen
- Steueranweisungen

Steuerstrukturen

- Folge (Sequenz)
- Auswahl (Selektion)
- Wiederholung (Zyklus)

Blöcke

- Zusammenfassung mehrerer Arbeitsschritte zu einer Anweisung höherer Ordnung
- Blockaufruf über Angabe des Blocknamens und Übergabe von Blockeingabeparametern
- Blockrückkehr mit Übergabe von Ergebnisparametern

Notation von Algorithmen

Verbal (Umgangssprache)

- einfache Realisierung ohne Spezialkenntnisse
- Beschränkung auf ein Sprachgebiet (z.B. Deutsch)
- Gefahr von Missverständnissen durch Mehrdeutigkeiten
- schwierige Prüfung der Korrektheit

Formal

- erfordert Spezialkenntnisse
- exakte, unzweideutige Notation
- Korrektheitsbeweis möglich
- international verständlich

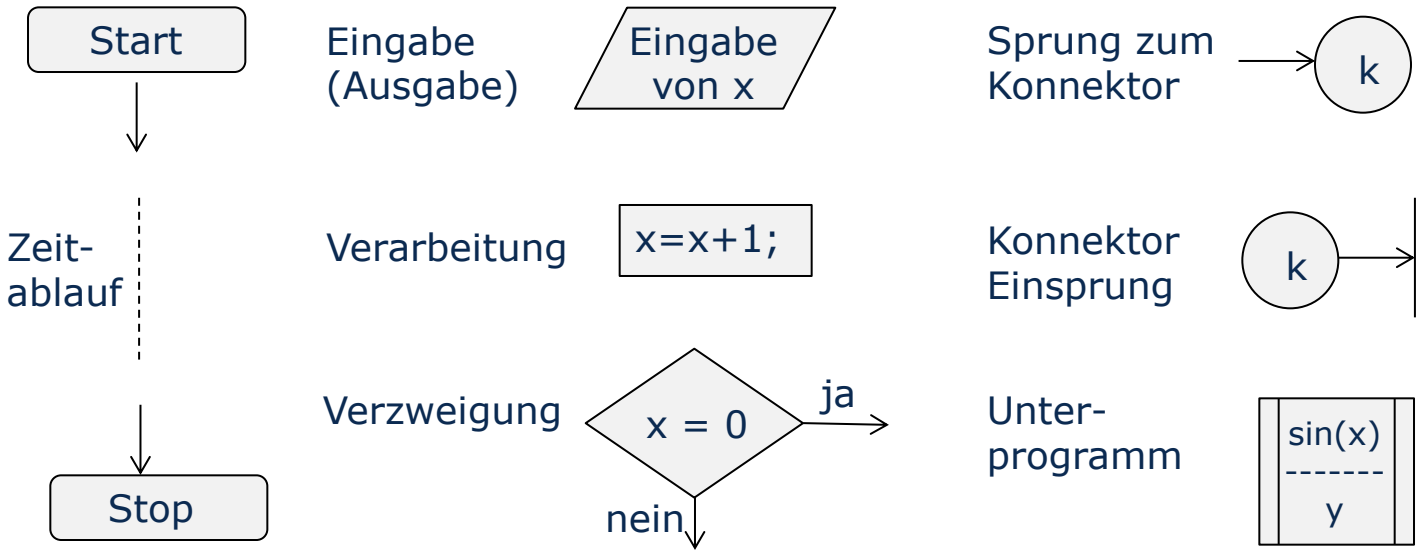
- Flussdiagramme (Programmablaufpläne)
- Struktogramme
- Pseudokode
- Programmiersprachen
- …

Programmablaufplan (PAP)

Notation von Algorithmen als Flussdiagramm (DIN 66001, ISO 5807)
(Pfeil symbolisiert Reihenfolge der Bearbeitungsschritte)

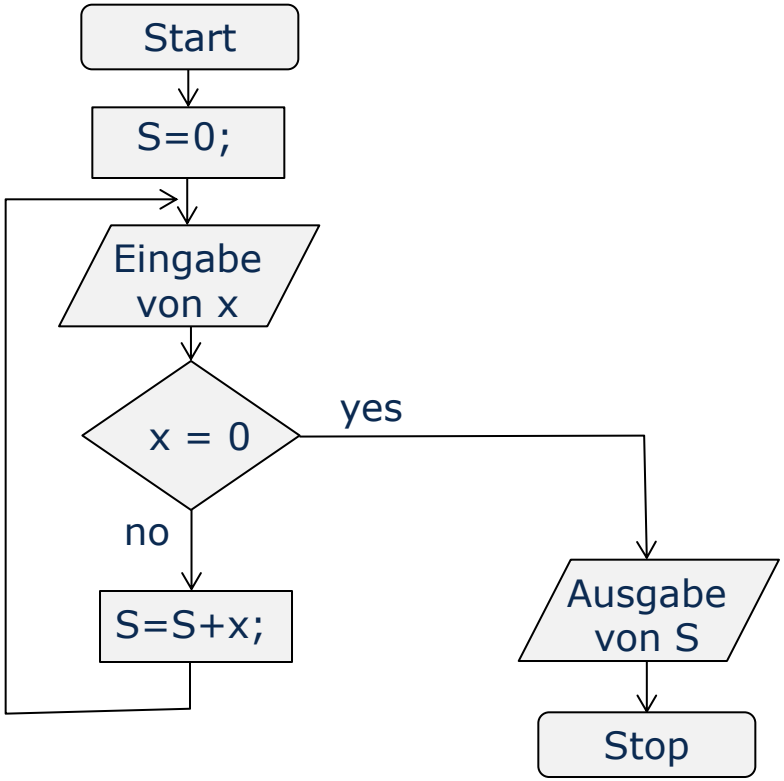
- einfache, anschauliche, grafische Darstellung,
- erlaubt aber auch Entwurf schlecht strukturierter Algorithmen

Sinnbilder (Auswahl)



PAP - Beispiele

Beispiel_1: „Summe reeller Zahlen; Abbruch bei Eingabe von 0“



Beispiel_2: www.arbeitsagentur.de/zentraler-Content/A06-Schaffung/A062-Beschaeftigungsverhaeltnisse/Publikation/pdf/PAP-KUG-2011.pdf

Struktogramme (Nassi-Shneidermann-Diagramme)

Grafische Notation von Algorithmen (DIN 66261, EN 28631), erzwingt den Entwurf gut strukturierter Algorithmen

Ein Struktogramm besteht aus Strukturblöcken

Erlaubt sind:

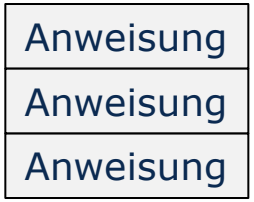
- sequentielle Folgen von Strukturblöcken

- geschachtelte Strukturblöcke

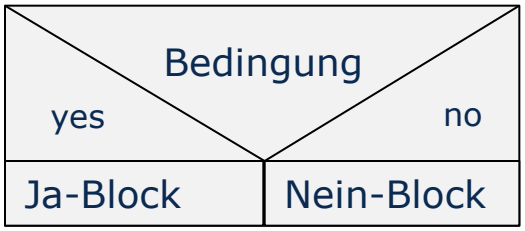
Struktogramme (Grundstrukturen)

Einzelblock

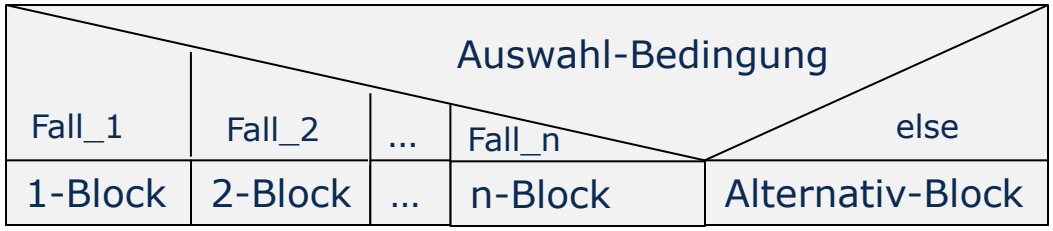
Sequenz-Block enthält eine Folge von Unterblöcken



Alternative umfasst zwei Blöcke



Mehrfachalternative umfasst … Blöcke

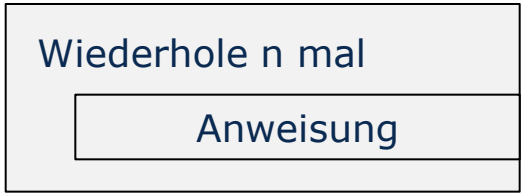


Struktogramme (Grundstrukturen)

Kopfgesteuerte Schleife (while)

Fußgesteuerte Schleife (do while)

Zählschleife (for)



evtl. vorzeitiges Schleifenende

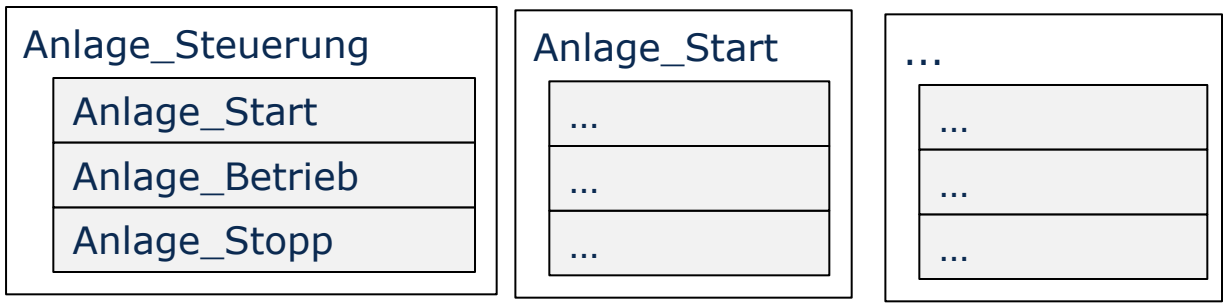
Struktogramme (Block-Hierarchien)

Struktogramme sollten nicht größer als max. eine A4-Seite sein.

→ Teile des Gesamtstruktogrammes auslagern
als separate Blöcke
(Darstellung nicht einheitlich geregelt)

Beispiel: „Steuerung einer Verkehrsanlage"

- sehr komplex, nicht in einem Struktogramm sinnvoll darstellbar, deshalb Unterteilung in 3 Sub-Blöcke
- Darstellung der Subblöcke in separaten Struktogrammen (Macros)
- evtl. weitere Unterteilung (top-down-Entwurf)



Struktogramme (Unterprogramme)

Struktogramme sollten kompakt sein.

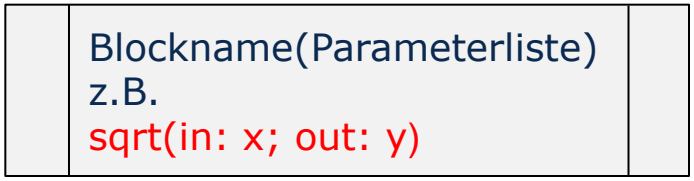
→ Mehrfach verwendete Teilstrukturen auslagern,
Unterprogramme in separaten Blöcken
(Darstellung nicht einheitlich geregelt)

Unterprogramm-Blöcke

- werden „aufgerufen", dabei i.a. Übergabe von Eingabeparametern
- danach arbeiten sie intern
- Abschließend erfolgt eine Rücksprung in den aufrufenden Block, i.a. mit Übergabe von Ergebnisparameter

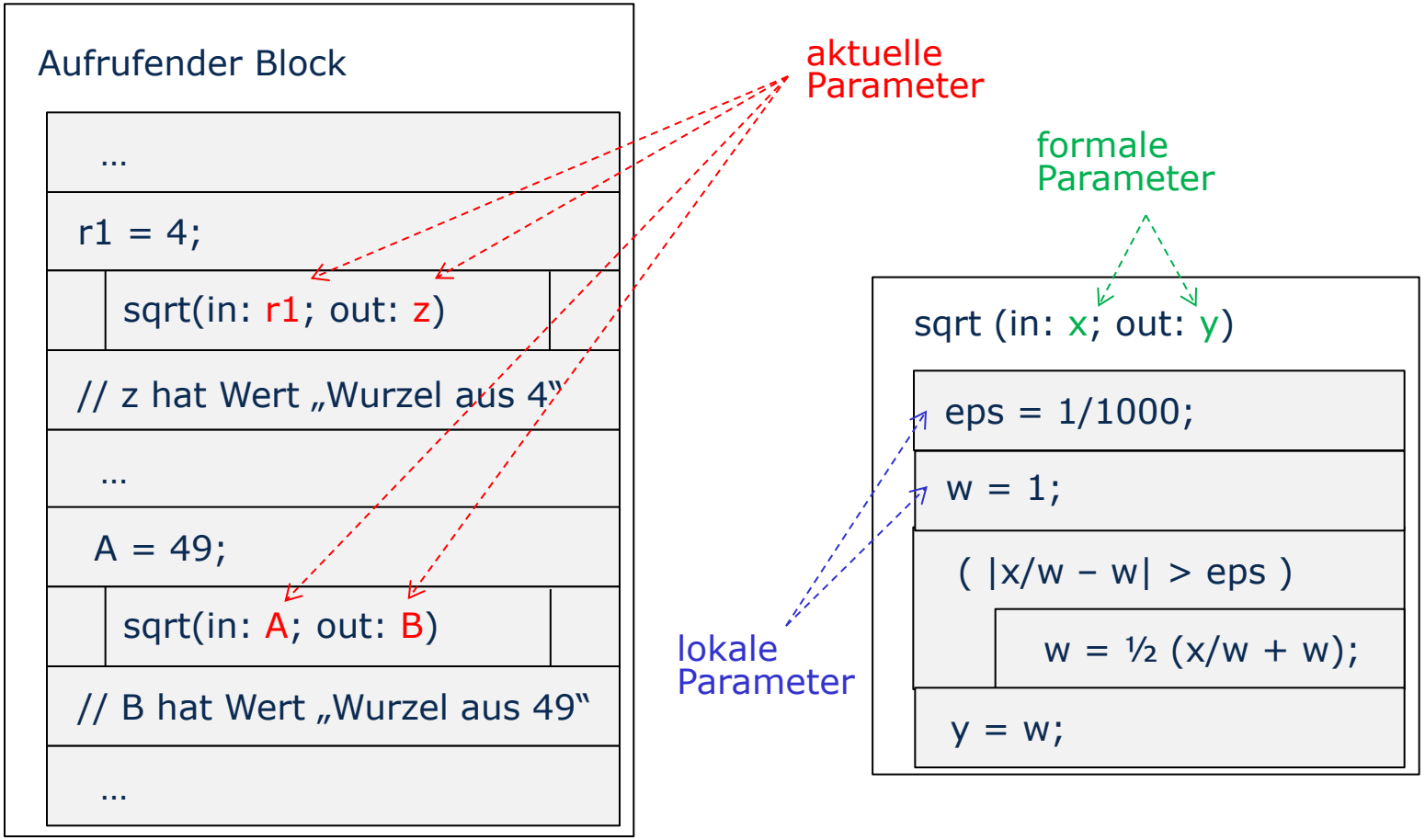
Beispiel: mathematische Funktion, wie y=sin(x), y=sqrt(x), …

Notation: Unterprogrammblock (nicht einheitlich standardisiert!)



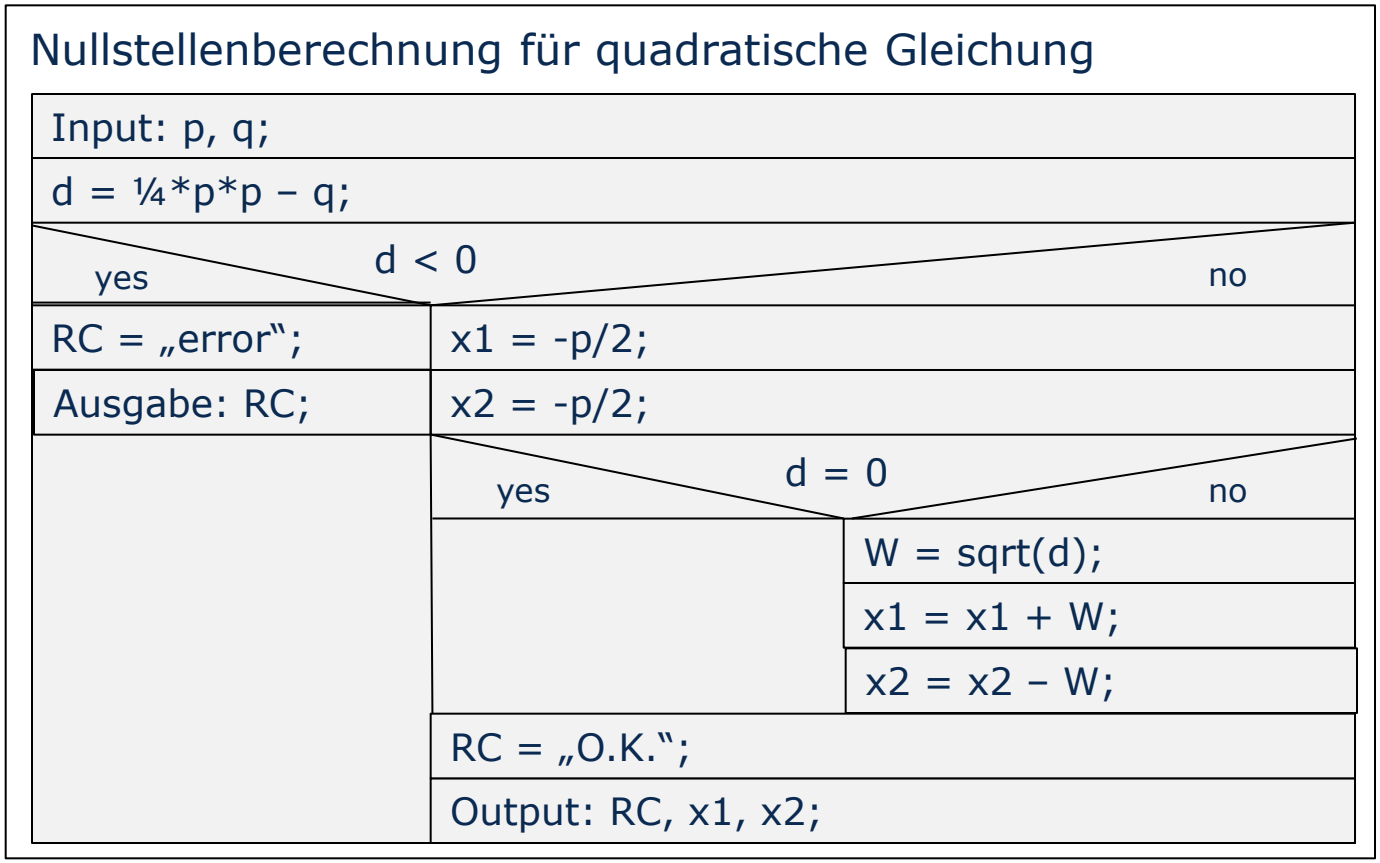
Struktogramme - Parameterübergabe

Beispiel: „mehrfacher Aufruf des Berechnens von Quadratwurzeln“



Nullstellen einer quadratischen Gleichung

$$x_{1/2} = -\frac{p}{2} \pm \sqrt{\left(\frac{p}{2}\right)^2 - q}$$ für $x^2 + p*x + q = 0$



Pseudokode

Text-Notation von Algorithmen

- nicht einheitlich standardisiert
- Anweisungen orientiert an einer Programmiersprache wie „C“ , „Pascal“, ...
- Anweisungen nicht für maschinelle Verarbeitung, nur für Verständnis des Autors
- halbverbal
  - zunächst weitgehend verbale Beschreibung

  danach
  - schrittweise Annäherung an Formulierung in Programmiersprache

Pseudokode - Beispiel

Größter gemeinsamer Teiler zweier natürlicher Zahlen
(Notation halbverbal)

```
ggt(a,b)

Input:      a ∈ N, b ∈ N
Output:     T ∈ N

while a ≠ b führe aus:

   if a > b then a = a - b else b = b – a

endwhile

T=a;
return T
```

Zahlenbeispiel

```
a=35   b=14

21     14
 7     14
 7      7

T=7
```

Daten

- besitzen einen **Typ**, gekennzeichnet durch
  - Wertebereich
  - zulässige Operationen
- besitzen zu jedem Zeitpunkt eine konkreten **Wert**.
- werden durch eine Zeichenfolge (**Bezeichner**) gekennzeichnet.
  - Bezeichner kennzeichnet Adresse des Datenspeicherplatzes.
  - Typ kennzeichnet Größe des Datenspeicherplatzes.
- werden innerhalb von Algorithmen durch Anweisungen bearbeitet.

z.B. `y=x;` // Wert der Variablen x wird
// umgespeichert auf den Platz der Variablen y

// Voraussetzung: gleicher Typ !!!

Datentypdefinition

vordefinierte Datentypen

- Abhängig von der gewählten Programmiersprache
  gibt es Datentypen, die bereits in der Sprache vordefiniert sind
  und deshalb nicht vom Programmierer definiert werden müssen.

- Beispiel-Typen (in "C"-Notation):

int	Typ Festkommazahl
float	Typ Gleitkommazahl

Individuelle Datentypdefinition

- Viele Sprachen erlauben die Definition eigener Datentypen.

- Beispiele:

  - Postanschrift
  - Kundencharakteristik
  - …

Datendeklaration

Festlegungen am Anfang eines Algorithmen-Blockes

für alle verwendeten Daten

- Zuordnung von Bezeichnern zu Datentypen
  Speicherreservierung im Programmkode (transparent)

- Pascal und verwandte Sprachen verlangen
  strenge und exakte Deklarationen ohne Ausnahmen.

- C und verwandte Sprachen sind toleranter.

Beispiele (in "C"-Notation)

```
int a,b;          // 2 ganze Zahlen im Bereich -32768…+32767
float x,y;        // 2 Gleitkommazahlen, 7-stellig
char ch='a';      // Zeichen (8 Bit), initialisiert mit ASCII-Zeichen
```

Datentypen

Idealisierte Datentypen

- z.B. mathematische Objekte,
  wie natürliche Zahlen, relle Zahlen,
- meist nicht im Computer realisierbar
  (unendlicher Wertebereich, Speicherplatz nicht planbar)

Konkrete Datentypen, realisierbar für Computer

- einfache Datentypen
- strukturierte Datentypen
- dynamische Variable, Zeiger

} Statische Typen

Abstrakte Datentypen (ADT)

- Verbergung von Implementierungsdetails,
  gut für „Programmierung im Großen“

ADT werden in dieser Lehrveranstaltungen nicht behandelt.

Einfache Datentypen (1)

integer ganze Zahlen

Wertebereich: 16 bit (short int) / 32 bit (int), 64 bit (long int)

Operationen: +, -, *, DIV (ganzahlige Division), >, <, …
(Bereichsüberschreitung möglich, z.B. bei Multiplikation)

char Zeichen

Wertebereich: i.a. 8 bit, Varianten beim Zeichensatz
z.B. ASCII, EBCDIC, …

Operationen: Manipulationen von Zeichenketten

boolean logisch „wahr“ bzw. „falsch“

Wertebereich: 1 bit, „0“ bzw. „1“

Operationen: Negieren, logisch UND bzw. ODER, …

Einfache Datentypen (2)

enum Aufzählung

Wertebereich: Menge von Eigenschaften, z.B. {Rot, Grün, …}

Operationen: Vergleich, Teilmengenbildung, …

real reelle Zahl

Darstellung: $\pm m*2^{exp}$ (Vorzeichen | Mantisse | Exponent)

Wertebereich: 32 bit (float) / 64 bit (double)

Operationen: +, -, *, /, >, <, …

(Bereichsüberschreitung möglich, z.B. bei Multiplikation
aber unwahrscheinlich,
da nur bei Exponentenüberlauf möglich)

Strukturierte Datentypen - Verbund

struct Verbund, bzw. Struktur oder Record

- fasst eine feste Anzahl von Daten (Komponenten) zusammen (unterschiedliche Typen möglich!)

Beispiel für Deklaration und Nutzung

```
// Typdefinition: (erfordert keinen Speicherplatz)

struct adresse { char Vorname[15]; char Name[15];
                 char Strasse[20]; char PLZ[5]; char Ort[15]; }

//

struct adresse Kunde1, Kunde2;          // Variablendeklaration
                                        // 2 Variable (140 byte)

Kunde1.Name = "Mustermann";             // Name für Kunde1 eintragen
```

Strukturierte Datentypen - Felder

array Feld

- fasst eine feste Anzahl von Daten gleichen Typs zusammen
- Komponenten werden über Indices angesprochen
- geeignet zur Darstellung von Vektoren und Matrizen

Beispiele für Deklaration und Nutzung

```
int zensur[24];        // Zensuren einer Schulklasse von 25 Schülern
                       // als eindimensionales Feld ganzer Zahlen
                       // (Index läuft ab Null)

zensur[2]=1;           // Schüler-3 erhält Note „1“


float temp[9][9][9]; // Temperaturen in jedem Kubikmeter
                      // eines Gebäudes als 3-dimensionales
                      // Feld von Gleitkommazahlen

T= temp[0][7][5];   // Temperatur in einem „Punkt“ des Gebäudes
```

Dynamische Variable

Pointer Zeiger

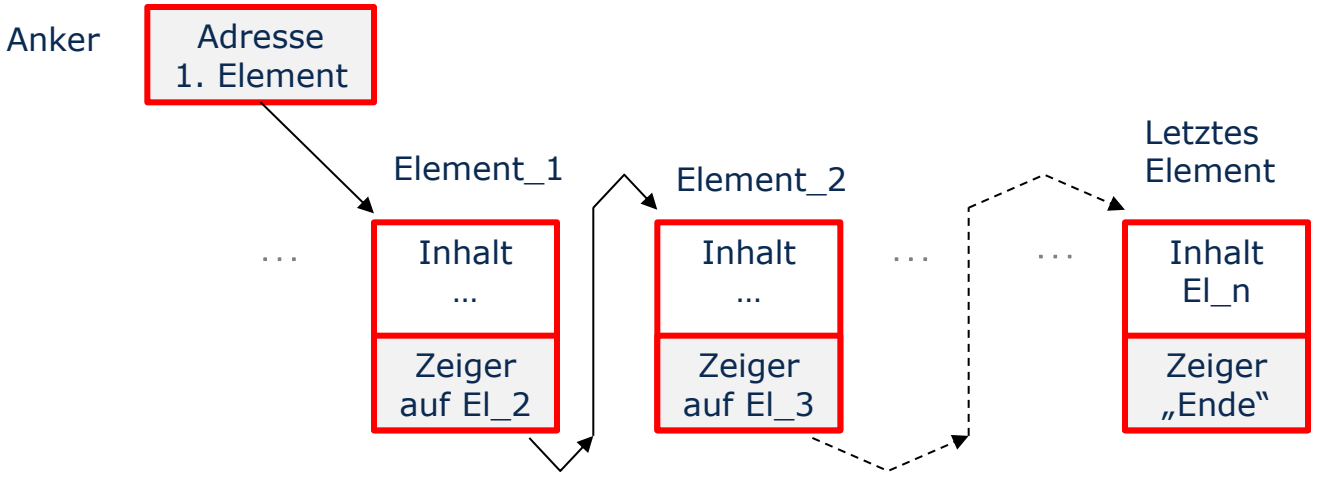
- ermöglichen Speicherverwaltung durch den Programmierer
- Zeiger-Variable haben als Wert eine Adresse.
- Operatoren (in „C“) & für „Adresse von“
  * für „Bereich der Adresse von“)

Beispiel:

```
int z;              // deklariert Variable z

int *zeiger;        // deklariert eine Zeigervariable

z=1;                // Der Wert 1 wird auf den Bereich von z kopiert.

zeiger = &z;        // Die Adresse von z wird kopiert
                    // auf den Bereich von zeiger

x = *zeiger         // Variable x bekommt den Wert von z
                    // (also den Wert 1)
```

lineare Listen

über Zeiger verkettete Speicherbereiche



Geeignet z.B. für

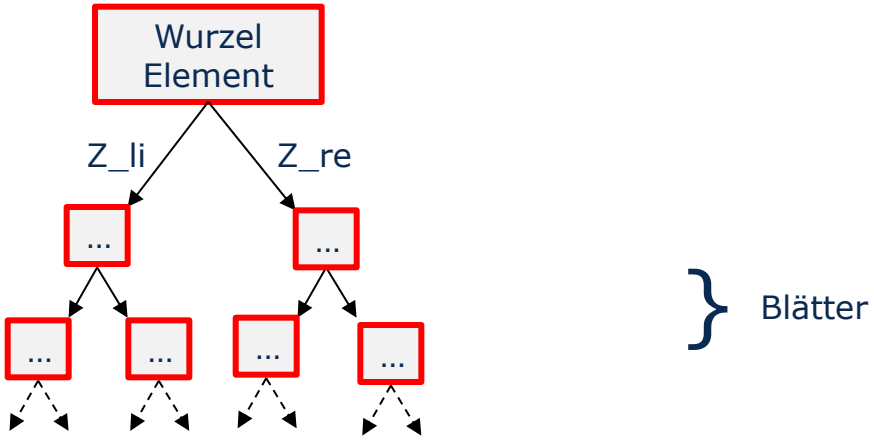
- Organisation von Warteschlangen
- Dateisystem FAT
- …

Bäume

Listen

bei denen die Speicherbereiche mehrere Verkettungszeiger besitzen,

z.B. **binäre Bäume** mit jeweils 2 Verzweigungsalternativen



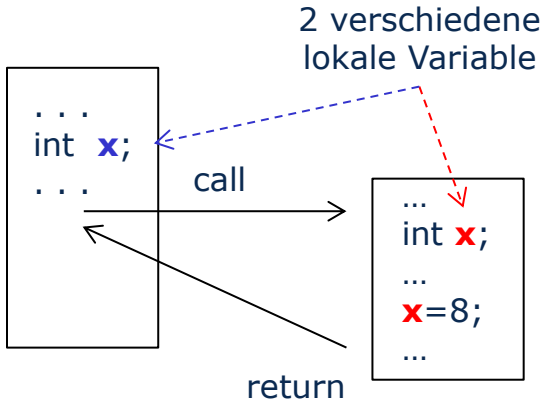
Geeignet z.B. für

- schnelle Sortierverfahren
- …

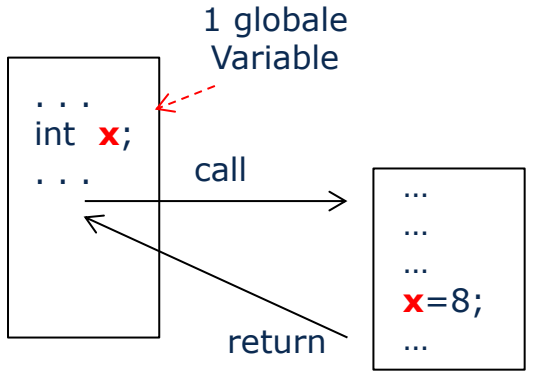
Lokale und Globale Daten

Programm normalerweise unterteilt in Haupt- und Unterprogramme

- Abbildung aktueller/formale Parameter beim Aufruf bzw. Rücksprung
- kein gegenseitiger Zugriff auf lokale Daten möglich ! (höchstens indirekt über Zeiger)



- deshalb Deklarierungsmöglichkeit für globale Daten
- Zugriff auf globale Daten ist im gesamten Programm möglich.



Modularisierung

Praktisch nie enthält ein Quellprogramm den gesamten Kode,
der für die Problemlösung benötigt wird.

- zusätzlich Nutzung
  fertig programmierter und bereits übersetzter Module, z.B.
  - Routinefunktionen der Compiler-Laufzeitumgebung
  - kommerziell erworbene Spezial-Pakete für Mathematik, …

- Problem: Wie kann ein Zugriff auf globale Daten erfolgen bei getrennter Übersetzung der Module?

- lösbar durch Angaben im Objektkode der einzelnen Moduln
  (Liste der globalen Variablen mit Namen und Relativadressen und Auflistung aller Befehle, die Globaladressen verwenden)

- Programmverbinder
  realisiert beim Binden der Module die Adressabbildungen.

Objektorientierte Programmierung (1)

konsequente Modularisierung

Objektklasse Objekt-Typdefinition

- Festlegung von Attributen (verwendete Datentypen)
- Festlegung von Methoden (zulässige Bearbeitungsprozeduren)
- Klassen können Eigenschaften anderer Klassen **erben** und erweitern.
  (Übernahme der Attribute und Methoden der vererbenden Klasse und Definition neuer Attribute und Methoden)
- Zur Laufzeit werden aus Klassen Objektinstanzen generiert, welche die eigentliche Programmarbeit durchführen.

Objektorientierte Programmierung (2)

Objekte autonome Bausteine eines Programmes

- starke **Abkapselung** gegenüber der Umgebung

  (innere Datenstruktur bleibt verborgen,
  Zugriff auf Daten nur über Aufrufe von Methoden)

Vor- und Nachteile der OOP

- klare Programmstrukturen, gute Übersichtlichkeit
- geringe Fehleranfälligkeit
- mitunter Effizienznachteile

Objektorientierte Programmiersprachen
z.B.

- C++
- C#
- Java → Vertiefung in Informatik-II