


**Folien zur Vorlesung  
Grundlagen systemnahes Programmieren  
Wintersemester 2016  
(Teil 2)**

**Prof. Dr. Franz Korf**

Franz.Korf@haw-hamburg.de

## Kapitel 2: Prozedurale Programmiersprachen

### Gliederung

- Einleitung 
- Datentypen
- Ausdrücke & Operatoren
- Kontrollstrukturen
- Strukturierung von Programmen: Funktionen, Unterprogramme & Module
- Zusammenfassung

Die Folien zu dieser Vorlesung basieren auf Ausarbeitungen von, Heiner Heitmann, Reinhard Baran und Andreas Meisel

## Paradigmen der Softwareentwicklung

- **Paradigmen** (Denkmuster) bestimmen unsere Vorstellung darüber, wie etwas funktioniert bzw. zu funktionieren hat.

Befehlssequenzen arbeiten auf Daten

Steuerung des Programmflusses durch bedingte/unbedingte Sprünge

Steuerung des Programmflusses durch strukturierte Elemente

Zusammenfassung von Befehlsfolgen zu Funktionen und Prozeduren

Zusammenfassung zusammengehöriger Daten+Prozeduren in Modulen

Datenorientierte Methodenentwicklung (Klassen, Vererbung, .....)

C++

C



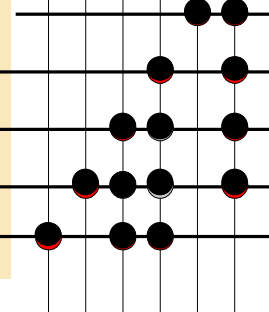
Maschinenorientierte Programmierung

Strukturierte Programmierung

Prozedurale Programmierung

Modulare Programmierung

Objektorientiert

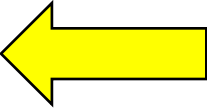


## Kennzeichen modularer/prozeduraler Sprachen

- einfache u. strukturierte Datentypen
  - Ausdrücke & Anweisungen
  - Kontrollstrukturen
  - Unterprogramme
  - Module
  - Gültigkeitsbereiche
- 
- Typische prozedurale Programmiersprachen: Fortran, COBOL, ALGOL, C, Pascal

## Kapitel 2: Prozedurale Programmiersprachen

### Gliederung

- Einleitung
- Datentypen 
- Ausdrücke & Operatoren
- Kontrollstrukturen
- Strukturierung von Programmen: Funktionen, Unterprogramme & Module
- Zusammenfassung

## Datentypen und Daten

**Datentypen** charakterisieren den Typ einer Information (Variablen) wie folgt:

- Wertebereich - welche Werte sie enthalten können
- welche Operationen auf sie angewendet werden können
- interne Darstellung im Speicher

Man unterscheidet:

- **einfache** Datentypen → speichern einzelne Werte (z.B. int)
- **strukturierte** Datentypen → speichern mehrere zusammengehörige Informationen (record, struct)
- **abstrakte** Datentypen → eigene Datentypen **mit Operationen** z.B. Stack (lifo), Queue (fifo)

Ein **Datenwort** (Datum) ist dann gekennzeichnet durch

- Datentyp
- Name
- Inhalt

## Einfache Datentypen

- Programmiersprachen definieren standardmäßig einen Satz von einfachen Datentypen.

Typ	Java	Wert
integer	int	2806
real	float, double	12.3
boolean	bool	true
character	char	'a'

- Moderne Sprachen erlauben eigene Definition einfacher Datentypen
  - Aufzählung von Konstantennamen
  - Java: `enum Ampel {ROT, GELB, GRUEN}`

## Strukturierte Datentypen

Typische strukturierte Datentypen prozeduraler Programmiersprachen sind

**array** *Ein -und mehrdimensionale Felder mit Komponenten des gleichen Datentyps. Die Komponenten werden über einen Index angesprochen.*

Quad	0		0 <-- Quad[0] 9 <-- Quad[3]
	1		
	4		
	9		
	16		

**record** *Datenverbund mit Komponenten verschiedenen Datentyps. Die einzelnen Komponenten werden über ihren Namen angesprochen.*

<b>Datentyp:</b> Kennzeichen			
Person	Alter	25	<b>DTyp:</b> int <b>DTyp:</b> Farbe <b>DTyp:</b> Farbe
	Haar	blond	
	Augen	blau	
			25 <-- Person.Alter

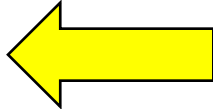
**file** *Sequenz von Komponenten gleichen Datentyps, wobei die Anzahl der Komponenten nicht festgelegt ist.*

**string** *Zeichenkette z.B.: "Das ist ein Text !!".*



## Kapitel 2: Prozedurale Programmiersprachen

### Gliederung

- Einleitung
- Datentypen
- Ausdrücke & Operatoren 
- Kontrollstrukturen
- Strukturierung von Programmen: Funktionen, Unterprogramme & Module
- Zusammenfassung

# Ausdrücke & Anweisungen

## Ausdrücke

- aufgeschriebene Formeln, die die Werte von Variablen und Konstanten miteinander verknüpfen und somit neue Werte berechnen.
  - Beispiele:  $a*(a+3)$   
 $x \text{ AND } (y \text{ OR } z)$

## Anweisungen

- **Zuweisungen** belegen die Inhalte von Variablen mit neuen Werten.
  - Beispiel:  $a = a * 4 + b$
- **Unterprogrammaufrufe** veranlassen die Ausführung von komplexeren Folgen von Anweisungen.
  - Beispiele: CopyString (Quellstring, Zielstring)  
Tag = BerechneWochentag(Datum)

## Operatoren (s. Java)

**Operator** sind Symbole, welches die Ausführung einer Aktion mit einem oder mehreren Ausdrücken festlegt.

- **unäre Operatoren** haben ein Argument (z.B. Negation, Inkrement)
- **binäre Operatoren** haben zwei Argumente (z.B. +, -, \*, /, AND, OR)

**Auswertungsreihenfolge:** Zur Auswertung komplexer Ausdrücke ist eine Auswertungsreihenfolge (Wertigkeit) für Operatoren definiert (z.B.:  $3 + 4 * 7$ ).

## Zuweisungen: lvalues und rvalues

### ➤ lvalue (l wie localizable)

- Wert, der an einem lokalisierbaren Ort gespeichert ist.
- Lokalisierbarer Ort: z. B. Speicher, Register
- Zuweisungen sind nur an lvalues möglich!

### ➤ rvalue (r wie readable)

- Wert, der aus der Auswertung eines beliebigen Ausdrucks entstehen kann

### ➤ Kommentar

- lvalue ist stets ein rvalue

### ➤ Beispiele (teilweise fehlerhaft)

`a = 3 + 4`

`3 = b` (*fehlerhaft*)

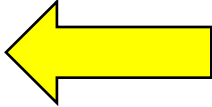
`b = a++ + 4`

`a+b = 7` (*fehlerhaft*)

`[a,b] = [3,7]` (Python)

## Kapitel 2: Prozedurale Programmiersprachen

### Gliederung

- Einleitung
- Datentypen
- Ausdrücke & Operatoren
- Kontrollstrukturen 
- Strukturierung von Programmen: Funktionen, Unterprogramme & Module
- Zusammenfassung

## Sequenz & Verbundanweisung (s. Java)

### Beispiel

*Sequenz* { `k = k - 1;`  
`if ( i > 0 )`  
    {  
        `i = i - 1;`  
        `erg = 2 * erg;`  
    }  
`j = j + 3;` }

*Verbundanweisung*

Eine Sequenz von Anweisungen,  
die in einem Block zusammen-  
gefasst sind

# Bedingte Anweisungen

Aus Java bekannt

- **Bedingte Anweisungen**
  - if-then-else Anweisung
  - case Anweisung

# Iteration

Aus Java bekannt

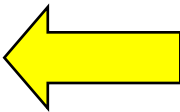
- **for** Schleife
  - Laufvariable, die an einen Bereich gebunden ist. Für jeden Wert des Bereichs wird die Schleife einmal durchlaufen.
- **kopfgesteuerte** Schleife
  - **while** Schleife
- **fußgesteuerte** Schleifen
  - Nach jedem Schleifendurchlauf wird ein boolescher Ausdruck ausgewertet.
  - Die Schleife wird mindestens einmal durchlaufen.
  - **repeat-until** Schleife: Ergibt der Ausdruck **true**, wird die Schleife beendet.
  - **do-while** Schleife: Ergibt der Ausdruck **false**, wird die Schleife beendet



## Kapitel 2: Prozedurale Programmiersprachen

### Gliederung

- Einleitung
- Datentypen
- Ausdrücke & Operatoren
- Kontrollstrukturen
- Strukturierung von Programmen: Funktionen, Unterprogramme & Module
- Zusammenfassung

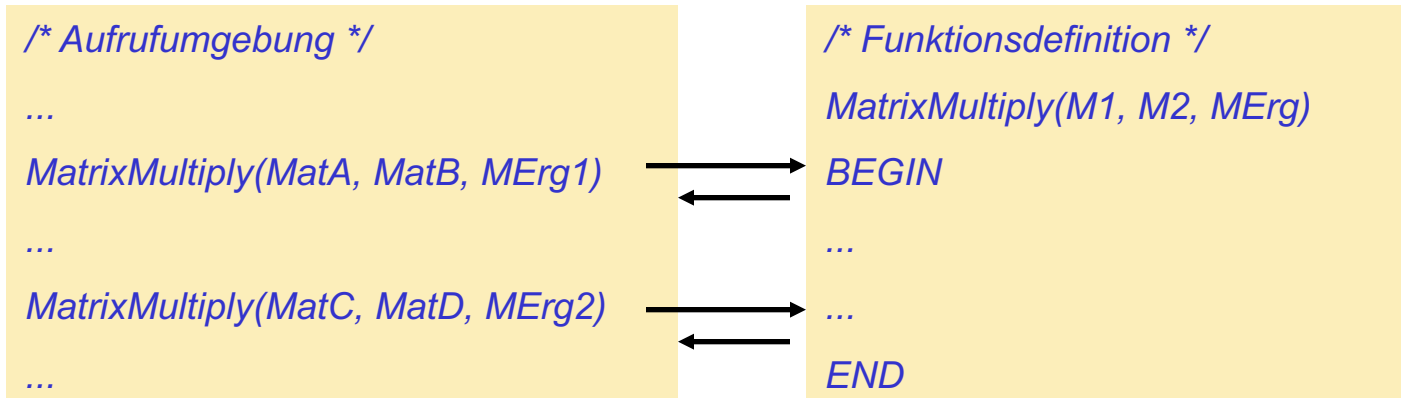


## Unterprogramme: Prozeduren und Funktionen

**Prozeduren und Funktionen** sind selbstdefinierbare Anweisungen.

- Sie sind die Träger von Algorithmen bzw. Berechnungen.
- Prozeduren werden durch Funktionen ohne Return Wert realisiert.
- Prozeduren und Funktionen haben
  - einen Namen, welcher den Zweck benennen sollte,
  - eigene (=lokale) Definitionen von Konstanten, Typen, Variablen,
  - Ein- und Ausgabeparameter
  - Neben den lokalen Variablen können sie auch gemeinsame (=globale) Variablen haben.

## Unterprogramme: Prozeduren und Funktionen (Fortsetzung)



➤ Unterschied zu Methoden in Java?

## Unterprogramme: Prozeduren und Funktionen (Fortsetzung)

### Parameterübergabemechanismen:

- Zur Einbindung der Unterprogramme in ihre Aufrufumgebung dienen die Parameter. Es gibt verschiedene Mechanismen der Parameterübergabe:
- **Call-by-Value:** Es werden die Werte (Kopien) übergeben.
- **Call-by-Reference:** Es wird eine Zugriffsmöglichkeit auf die Speicherplätze übergeben, die beim Aufruf des Unterprogramms in der Parameterliste auftreten.  
Somit kann das Unterprogramm die Werte dieser Speicherplätze modifizieren.

## Parameterübergabemechanismen in C

- In C sind die Funktionsparameter **Call-by-Value** Parameter

```
void f1(int v) {  
    v = 9;  
}  
  
...  
int test = 0;  
f1(test); /* Welchen Wert hat test nach dem Aufruf? */
```

- **Call-by-Reference:** Parameter werden über indirekte Adressierung – Pointer realisiert.

**Die Adresse des Objektes wird als Call-by-Value Parameter übergeben.**

```
void f1(int* v) {  
    *v = 9;  
}  
  
...  
int test = 0;  
f1(&test); /* Welchen Wert hat test nach dem Aufruf? */
```

**int \*** : Definiert den Typ "Adresse (Pointer, Zeiger) auf ein int Objekt"

**\*** liefert den Speicherplatz mit der Adresse, die v speichert.

## Beispiel Parameterübergabe

```
int a = 6, b = 9, c = 10, d = 11;
void swap_1(int x, int y){
    int t = x;
    x = y;
    y = t;
}
void swap_2(int* x, int* y){
    int t = *x;
    *x = *y;
    *y = t;
}
swap_1(a,b);
swap_2(&c,&d);
```

& liefert die Adresse einer Variablen, den Pointer auf eine Variable.

Welche Werte haben a, c, c, d nach der Ausführung der Funktionen?

a	b	c	d
6	9	11	10

# Funktionsdeklaration vs. Funktionsdefinition

## Unterschied: Deklaration – Definition

**Deklaration** = Bekanntgabe von **Name** und **Typ** eines Objektes an den Compiler.

**Funktionsdeklaration**: Dem Compiler werden nur Name, Parameterliste und Ergebnistyp der Funktion „mitgeteilt“ – der Funktionskopf enthält genau diese Informationen (= Signatur der Funktion).

**Definition** = Detailbeschreibung eines Objektes.

**Funktionsdefinition**: Funktionskopf und **Rumpf** der Funktion.

## Beispiel

```
/* Variablendefinitionen */
int i, j, k;

/* Funktionsdeklaration */
int QuadSum (int x, int y);

int main() {
    ...
    k=QuadSum(m,n); /*Aufruf*/
    ...
}
```

```
/* Funktionsdefinition */
int QuadSum (int x, int y)
{
    int res;
    res = x * x + y * y;
    return res;
}
```

## Seiteneffekte

**Seiteneffekt:** Wenn eine Funktion den Wert einer globalen Variablen verändert, liegt ein Seiteneffekt vor.

Allein der Aufruf solch einer Funktion kann zu einer Veränderung des Systems führen – auch wenn der Rückgabewert nicht weiter verarbeitet wird.

**Beispiel:**

```
int counter = 0;
int f(x) {
    counter = counter + 1;
    return x*x;
}
```

**Diskussion:** Relation zu set Funktionen  
Relation zu Methoden von Klassen

**Diskussion** lazy evaluation of boolean expressions

```
if ((x == y) && (4 == f(2))) ...
```

- Der Ausdruck wird nur soweit ausgewertet, bis ein eindeutiges Ergebnis vorliegt. Wird f(2) stets aufgerufen?
- Auswertungsreihenfolge des booleschen Ausdrucks ist entscheidend



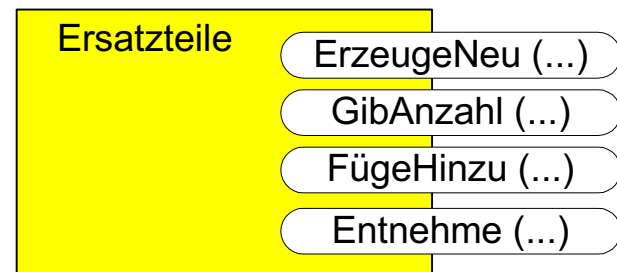
## Module

Größere Programme werden aus Gründen der Übersichtlichkeit, Wiederverwendbarkeit und arbeitsteiligen Erstellung in mehrere Einheiten aufgeteilt.

**Modul** = Sammlung von Daten und Programmen, die diese Daten verarbeiten.

- Ein (gutes) Modul hat einen zentralen Zweck.
- Ein (gutes) Modul verbirgt sein Innenleben (information hiding).
- Ein gutes Modul bietet seine Funktionalität (= Zugriffe und Manipulationen seiner Daten) über eine Schnittstelle an.
- Ein (gutes) Modul ist vollständig in dem Sinne, dass keine anderen Module direkt auf seine Daten zugreifen.

Beispiel: Modul zur Verwaltung von Ersatzteilen



***Ein gutes Modulkonzept ist entscheidend für die Qualität eines komplexen Programmsystems.***

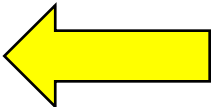
## Module (Fortsetzung)

Typische Situationen, wenn ein Modul angelegt werden soll:

- Trennung von Benutzerschnittstelle und Funktionalität
- Vermeidung von Hardware-Abhängigkeiten
- Ein-/Ausgabefunktionen
- Kapselung (von Systemabhängigkeiten)
- abstrakte Datentypen
- Bündelung von wieder verwendbarem Code
- ...

## Kapitel 2: Prozedurale Programmiersprachen

### Gliederung

- Einleitung
- Datentypen
- Ausdrücke & Operatoren
- Kontrollstrukturen
- Strukturierung von Programmen: Funktionen, Unterprogramme & Module
- Zusammenfassung 

## Zusammenfassung

- Paradigmen
- Typische Eigenschaften prozeduraler/modularer Programmiersprachen
- Grundlegende Konzepte im Schnelldurchlauf:
  - Einfache und strukturierte Datentypen
  - Typische Ausdrücke, Operatoren und Kontrollstrukturen
  - Funktionen und Unterprogramme als Strukturierungsmittel
  - Call-By-Value & Call-By-Reference
  - Module