

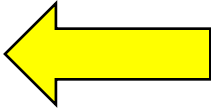
**Folien zur Vorlesung  
Grundlagen systemnahes Programmieren  
Sommersemester 2016  
(Teil 2)**

**Prof. Dr. Franz Korf**

Franz.Korf@haw-hamburg.de

## Kapitel 2: Produrale Programmiersprachen

### Gliederung


- Steckbriefe 
- Formalien und Kommentare
- Inhalt der Vorlesung
- Zusammenfassung

### Die Folien zu dieser Vorlesung basieren auf Ausarbeitungen von

- Heiner Heitmann
- Reinhard Baran
- Andreas Meisel

## 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



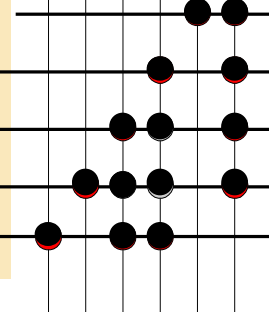
Maschinenorientierte Programmierung

Strukturierte Programmierung

Prozedurale Programmierung

Modulare Programmierung

Objektorientiert

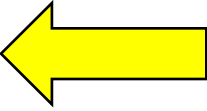


## Kennzeichen modularer/prozeduraler Sprachen

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

## Kapitel 3: 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. Dies sind üblicherweise

- **integer** Zahlendarstellung von natürlichen/ganzen Zahlen (12, 5, -34)
- **real** Darstellung von reellen Zahlen (12.34, 12.56E6, )
- **char** Darstellung von alphanum. Zeichen ('a')
- **boolean** Darstellung logischer Werte TRUE und FALSE

Viele Sprachen verfolgen bezüglich der Datentypen das Konzept des **Strong Typing**, d.h. einer Variablen können nur Werte des vereinbarten Typs zugewiesen werden.

Moderne Sprachen erlauben eigene Definition einfacher Datentypen

- **enum** Aufzählung von Konstantennamen ( Ampel = {rot, gelb, grün} )



## 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.*

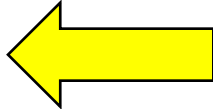
<b>Datentyp</b> Kennzeichen			
Person	Alter	25	25 <-- Person.Alter
	Haar	blond	
	Augen	blau	

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

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

## Kapitel 3: 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

**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)

Typische Kategorien von Operatoren:

- Arithmetische Operatoren (+, -, \*, /, ...)
- Zuweisungsoperatoren (=)
- Logische Operatoren (AND, OR, NOT, ...)
- Bitweise Operatoren (bitw. AND, bitw. OR, ...)
- Vergleichsoperatoren (EQ, LT, GT, ....)
- Zeichenketten-Operatoren

**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)

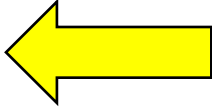
```
a = 3 + 4  
a+b = 7
```

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

```
b = a++ + 4
```

## Kapitel 3: Prozedurale Programmiersprachen

### Gliederung

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

# Sequenz & Verbundanweisung

## Sequenz

- Eine Folge von Anweisungen und Kontrollstrukturen, die hintereinander ausgeführt werden.

## Verbundanweisung

- Eine Sequenz von Anweisungen, die in einem Block zusammengefasst sind.

## Beispiel

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

## Bedingte Anweisungen: If & Case Anweisung

### If Anweisung

- In Abhängigkeit des Ergebnisses eines booleschen Ausdrucks wird eine Anweisungsfolge (oder eine alternative Anweisungsfolge) ausgeführt.
- **if** (Boolescher Ausdruck)  
    **then**  
        Sequenz von Anweisungen  
    **end if**
- **if** (Boolescher Ausdruck)  
    **then**  
        Sequenz von Anweisungen  
    **else**  
        Sequenz von Anweisungen  
    **end if**



## Bedingte Anweisungen: If & Case Anweisung (Fortsetzung)

### Case Anweisung

- In Abhängigkeit des Ergebnisses eines Ausdrucks wird eine von mehreren alternativen Anweisungsfolge ausgeführt.
- **switch** Ausdruck **of**
  - case** Konstante, Konstante, ... :  
Sequenz von Anweisungen
  - case** Konstante, Konstante, ... :  
Sequenz von Anweisungen
  - ...
  - default:**  
Sequenz von Anweisungen

}

*optional*

- end switch**

## Iteration: While Schleife

### While Schleife (kopfgesteuerte Schleife)

- Vor jedem Schleifendurchlauf wird ein boolescher Ausdruck ausgewertet. Ist dieser TRUE, wird der Schleifenrumpf ausgeführt. Anschließend wird erneut anhand des booleschen Ausdruck überprüft, ob die Schleife nochmals durchlaufen werden soll.  
Liefert die Auswertung des booleschen Ausdrucks FALSE wird die Programmausführung hinter der Schleife fortgesetzt.
- **while** boolescher Ausdruck **do**  
    Sequenz von Anweisungen  
**end while**

## Iteration: Repeat Schleife (Fortsetzung)

### Repeat Schleife (fußgesteuerte Schleife)

- **Nach** jedem Schleifendurchlauf wird ein boolescher Ausdruck ausgewertet. Ist dieser FALSE, wird der Schleifenrumpf nochmals ausgeführt. Liefert die Auswertung des booleschen Ausdrucks TRUE, wird die Programmausführung hinter der Schleife fortgesetzt.
- Der Schleifenrumpf wird mindestens einmal durchlaufen.
- Zentraler Unterschied zur do while Schleife aus C: Die do while Schleife wird erneut durchlaufen, wenn der boolesche Ausdruck TRUE ist.
- **repeat**  
Sequenz von Anweisungen  
**until** boolescher Ausdruck **end repeat**

## Iteration: For Schleife (Fortsetzung)

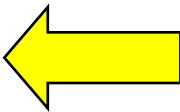
### For Schleife

- Eine Laufvariable ist an die for Schleife gebunden. Diese Variable durchläuft einen im Schleifenkopf festgelegten Bereich. Für jeden Wert aus diesem Bereich wird die Schleife einmal durchlaufen.
- **for** Laufvariable **in** Anfangswert **to** Endwert **do**  
Sequenz von Anweisungen  
**end for**

## Kapitel 3: Prozedurale Programmiersprachen

### Gliederung

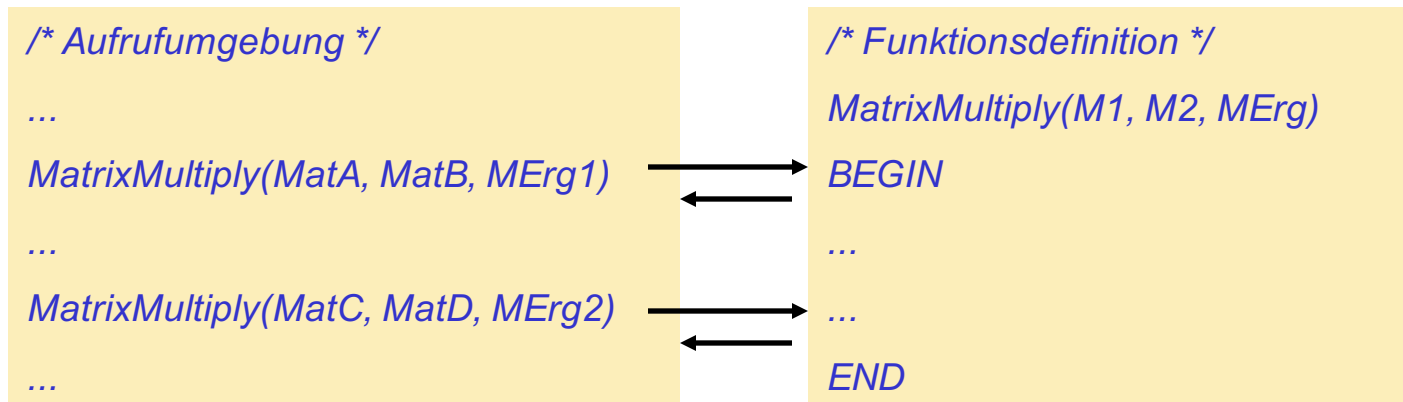
- Einleitung
- Datentypen
- Ausdrücke & Operatoren
- Kontrollstrukturen
- Strukturierung von Programmen: Funktionen, Unterprogramme & Module
- Zusammenfassung



## Unterprogramme und Funktionen

**Unterprogramme** sind selbstdefinierbare Anweisungen. Sie 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 mit der Aufrufumgebung haben.



## Unterprogramme 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.

Funktionen sind Unterprogramme, die wie mathematische Funktionen ein Ergebnis liefern.

## Funktionsdefinition und C (kleine Einführung)

- Funktionen sind die Träger von Algorithmen bzw. Berechnungen.
- Prozeduren werden durch Funktionen ohne Return Wert realisiert.

### Beispiele:

```
void wenig (void ) { ... }
```

```
int inkrementiere ( int zahl, int incr ) {
```

```
    int localvar = 66;
```

```
    ....
```

```
}
```

```
void update ( float druck, int temp, int wert ) { ... }
```



## Funktionsdefinitionen (Fortsetzung)

### 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.

**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;
}
```

## return Anweisung

Die **return** Anweisung beendet die Ausführung einer Funktion und gibt, sofern für die Funktion ein Ergebnistyp vereinbart wurde, das berechnete Ergebnis an seine Aufrufumgebung zurück.

### Syntax:

`return_statement ::= "return" [ expression ] .`

Der Wert des optionalen Ausdrucks liefert den Return-Wert der Funktion und muss zum Ergebnistyp der Funktion kompatibel sein.

### Beispiel:

```
return ( a + b ) / 2 ;
```

## 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? */
```

## 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

**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

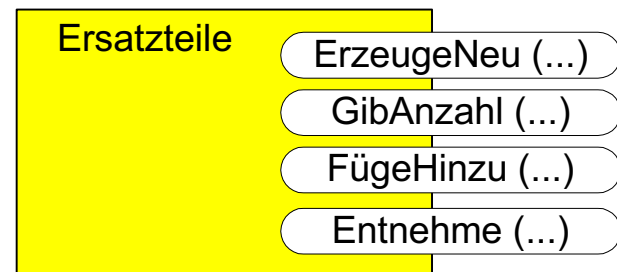
## Module

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.***

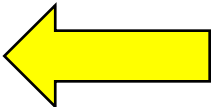
## Module (Fortsetzung)

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 3: 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