

Praxis der Programmierung

Funktionen, Header-Dateien, Pointer

**Institut für Informatik und Computational Science
Universität Potsdam**

Henning Bordihn

Organisatorische Bemerkungen

Modul Programmierung

- Pflichtmodul für BSc INF (alte Ordnung) und BSc WIN
- **1. Fall:** Sie studieren im 2. Fachsemester.

Praxis der Programmierung ersetzt die Programmierung.

Modul Programmierung

- Pflichtmodul für BSc INF (alte Ordnung) und BSc WIN

- **1. Fall:** Sie studieren im 2. Fachsemester.

Praxis der Programmierung ersetzt die Programmierung.

- **2. Fall:** Sie studieren im 4. Fachsemester oder höher.

Die bisherige Programmierung wird nicht mehr angeboten.

Sie können im Juli aber eine Klausur schreiben und das Modul abschließen.

Modul Programmierung

- Pflichtmodul für BSc INF (alte Ordnung) und BSc WIN
- **1. Fall:** Sie studieren im 2. Fachsemester.
Praxis der Programmierung ersetzt die Programmierung.
- **2. Fall:** Sie studieren im 4. Fachsemester oder höher.
*Die bisherige Programmierung wird nicht mehr angeboten.
Sie können im Juli aber eine Klausur schreiben und das Modul abschließen.*
- **Ausnahme:** Sie studieren im 4. Fachsemester oder höher,
haben aber *Einführung in die Programmierung* (2. Teil von RNB) nie belegt.
Praxis der Programmierung ersetzt die Programmierung.

Funktionen in C

Definition von Funktionen

Funktionskopf Funktionsrumpf

1. Funktionskopf (Signatur)

- *Rückgabetyt Bezeichner (Typ_1 formaler_Parameter_1, Typ_2 formaler_Parameter_2, \vdots Typ_n formaler_Parameter_n)*
- legt Namen der Funktion fest: *Bezeichner*
- legt Definitionsbereich der Funktion fest: Liste *formaler Parameter*
- legt Wertebereich der Funktion fest: *Rückgabetyt*
- legt fest, wie die Funktion aufgerufen wird

2. Funktionsrumpf (Implementierung)

- ist ein Block (Sequenz von Anweisungen)
- legt fest, wie die Eingabewerte verarbeitet werden
- return-Anweisung \rightsquigarrow Rückkehr zu aufrufender Funktion
 \rightsquigarrow Rückgabewert übergeben

Beispiel:

```
int quadrat (int n) {  
    return n * n;  
}
```

\rightsquigarrow nach return kann Ausdruck stehen

Erinnerung: Blöcke und Variablen

- In Blöcken definierte Variablen sind nur innerhalb dieses Blockes sichtbar.
 - ~> auch in enthaltenen Blöcken
 - ~> aber **nicht** in umfassenden Blöcken
- In Blöcken definierte Variablen *verdecken* gleichnamige Variablen von umfassenden Blöcken.
- In Blöcken definierte Variablen werden beim Verlassen des Blockes wieder ungültig.
 - ~> überdeckte Variablen werden wieder sichtbar
- Funktionsrümpfe sind Blöcke!!! Lokal sind:
 - Variablen, die im Rumpf definiert werden,
 - Parameter der Funktion

Funktionsaufruf

Funktionsname (aktueller_Parameter_1, ..., aktueller_Parameter_n);

1. automatisches Anlegen von lokalen Variablen für die Parameter

\rightsquigarrow *typ_i formaler_Parameter_i;*

2. automatische Initialisierung mit aktuellen Parametern

\rightsquigarrow *formaler_Parameter_i = aktueller_Parameter_i;*

Aktuelle Parameter können **Ausdrücke** sein!

3. Abarbeitung der Anweisungen im Funktionsrumpf

Funktionsaufruf ist Ausdruck

```
int qud = quadrat(12);      // liefert qud = 144;
```

Rückgabety `void`

- Funktionen ohne Rückgabewert
 \rightsquigarrow reine Prozeduren
- dienen z.B. zur bloßen Datenausgabe
- keine `return`-Anweisung *oder* `return`;
- `void prozedur_funktion (...) { ... }`

Deklarieren *versus* Definieren

- Funktion deklarieren

Festlegung der Schnittstelle (Signatur)

```
long cube (int n);
```

- Funktion definieren

Festlegung der Schnittstelle (Signatur) und des Verhaltens (Implementierung)

```
long cube (int n) {  
    return n*n*n;  
}
```

- Anwendung: **Vorwärtsdeklaration**

Funktionen müssen *vor* ihrem ersten Aufruf deklariert sein
(Implementierung kann an anderer Stelle erfolgen)

Vorwärtsdeklaration

- Beispiel:

```
void g(int n);  
void f(int n) { g(n-1); }  
void g(int n) { f(n-1); }
```

- Schlüsselwort `extern` zeigt an, dass die Definition an anderer Stelle in derselben oder einer anderen Datei erfolgt

```
extern int value;    // Definition erfolgt an anderer Stelle  
...  
int value = 2014;    // muss vollstaendige Definition sein
```

- vorwärtsdeklarierte Funktionen sind implizit `extern`

```
void g(int n);          // steht fuer extern void g(int n);
```

Header-Dateien

Definition von Header-Dateien

- *dateiname.h*
- Header-Dateien enthalten (meist) nur Deklarationen (Signaturen)
- Die Definitionen (Implementierungen) sind (meist) in .c-Dateien enthalten.

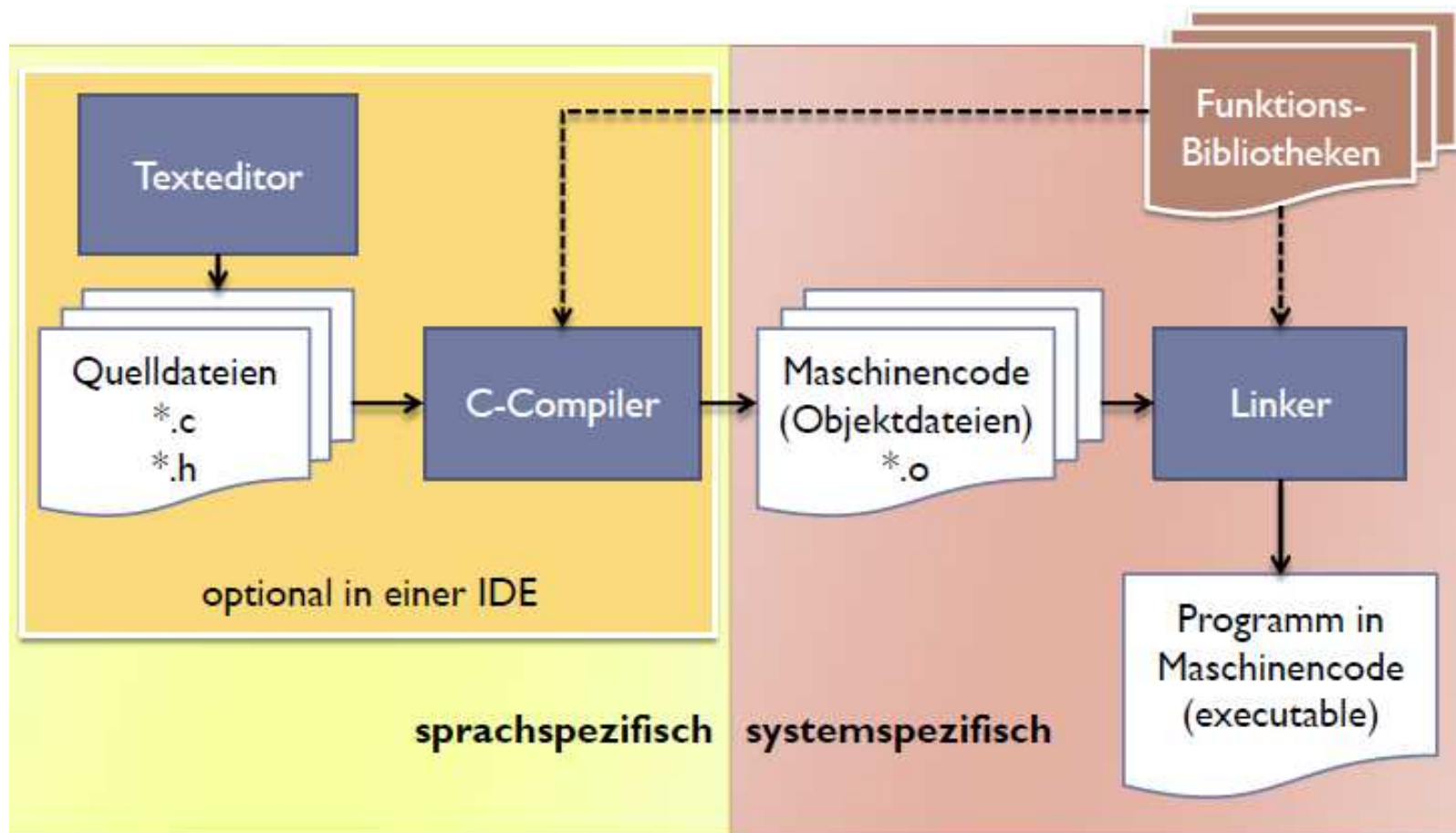
- Beispiel bisher:

```
void g(int n);  
void f(int n) { g(n-1); }  
void g(int n) { f(n-1); }
```

- mit Header-Datei:

```
#include "bsp.h"                // enthaelt extern void g(int n);  
void f(int n) { g(n-1); }  
void g(int n) { f(n-1); }
```

Einbinden der Header-Dateien (1)



Einbinden der Header-Dateien (2)

- Präprozessor setzt diese an der Stelle der `#include`-Direktive ein
 \rightsquigarrow Funktionen sind dem Compiler bekannt
- Implementierung erst für das Laufzeitsystem zum Zeitpunkt des Funktionsaufrufs bedeutsam
- Linker fügt die Objektcode-Dateien zusammen
 - ggf. verschiedene C-Dateien und Bibliotheksdateien
 - Objektcode-Dateien verwenden relative/virtuelle Speicheradressen
 - Linker sorgt für einheitlichen Adressraum des ausführbaren Programms (Zuordnung eines virtuellen, überschneidungsfreien Adressraums)

Fehlerarten

- **Compilerfehler:** Fehler, die der Compiler erkennt, z.B.
 - Semikolon vergessen
 - Variable benutzt aber nicht definiert etc.

⇒ meist kein ausführbares Programm
- **Laufzeifehler:** Fehler, die bei der Abarbeitung des Programms auftreten, z.B.
 - Division durch 0
 - fehlende Werte
 - Zugriff auf Datei, die nicht geöffnet werden kann
 - kein Speicherplatz mehr vorhanden etc.
- **logische Fehler:** Fehler im Programmentwurf

Pointer (Zeiger)

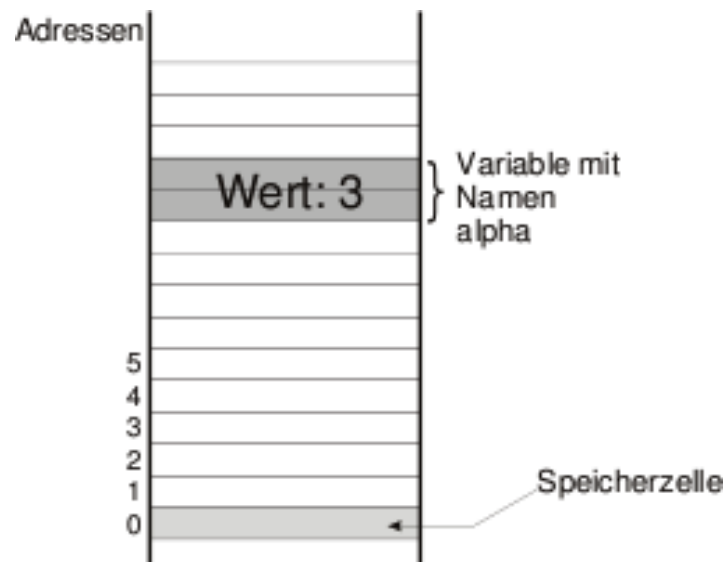
Vier Kennzeichen einer Variablen

- Datentyp
- Variablenname
- Wert
- Adresse im Arbeitsspeicher (Primärspeicher)

In **C**: Zugriff über Namen oder Adresse auf den Wert

Adressen von Variablen

- Arbeitsspeicher ist in (gleich große) *Speicherzellen* eingeteilt (z.B. je 1 Byte)
- Speicherzellen sind durchnummeriert (Hexadezimalzahlen)
- **Adresse** einer Variablen ist *Nummer* der Speicherzelle, in der ihr Speicherplatz beginnt.



Variablen und ihre Adressen

Variable v

speichert den Wert eines bestimmten Datentyps an einer Speicheradresse

Deklaration: `int v;`

v hat die Adresse $\&v$

$\&$ - **Adressoperator**

ermittelt die Adresse / die Referenz (den Pointer) auf v

Variablen und ihre Adressen

Variable v

speichert den Wert eines bestimmten Datentyps an einer Speicheradresse

Deklaration: `int v;`

v hat die Adresse $\&v$

$\&$ - Adressoperator

ermittelt die Adresse / die Referenz (den Pointer) auf v

Pointer p

speichert die Adresse von v , an der ihr Wert abgelegt wird

Deklaration: `int *p;`

p *zeigt* auf Speicherplatz mit Wert $*p$

$*$ - Inhaltsoperator

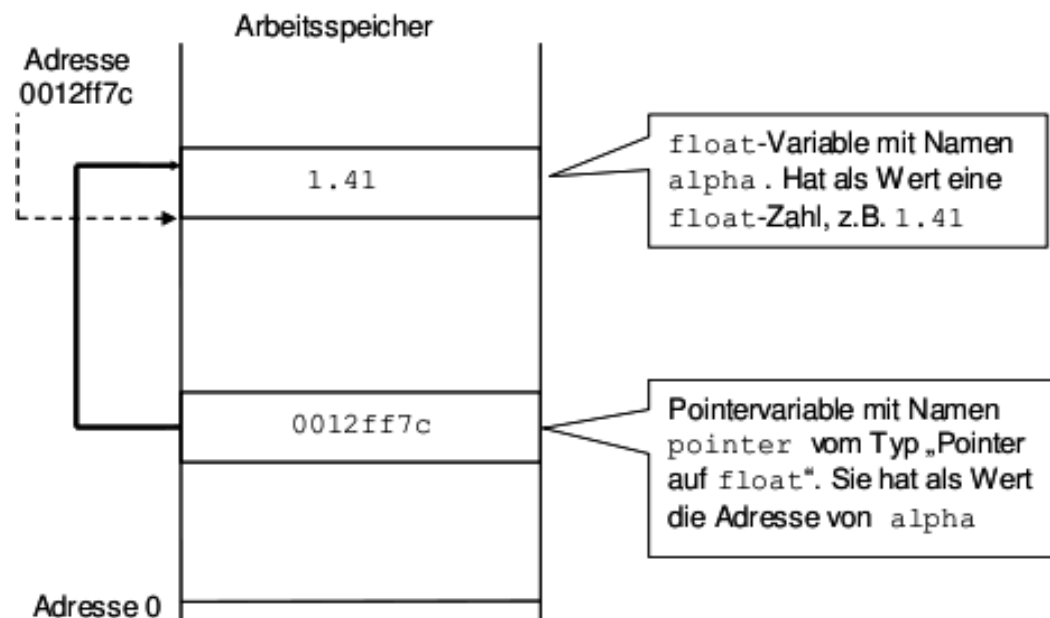
ermittelt den Wert an Speicherplatz p (Dereferenzierungsoperator)

Pointer

- **Pointer:** ist Variable

Typ: Pointer auf einen bestimmten Datentyp (z.B. float)

Wert: *Adresse* einer Variablen



Pointervariablen

- Definition als Pointer auf *Datentyp*
~> Pointertyp und Datentyp des Speicherobjekts sind gekoppelt!
- *Typname * Pointername;*
~> Datentyp: Pointer auf *Typname*
- Beispiel: `float * pointer1;`
~> Pointer auf `float` mit Namen `pointer1`

Pointervariablen

- Definition als Pointer auf *Datentyp*
 \rightsquigarrow Pointertyp und Datentyp des Speicherobjekts sind gekoppelt!
- *Typname* * *Pointername*;
 \rightsquigarrow Datentyp: Pointer auf *Typname*
- Beispiel: `float * pointer1`;
 \rightsquigarrow Pointer auf `float` mit Namen `pointer1`

• <u>Vorsicht</u> :	Definition	entspricht
	<code>int * pointer, alpha;</code>	<code>int * pointer;</code> <code>int alpha;</code>
	<code>int * pointer1, * pointer2;</code>	<code>int * pointer1;</code> <code>int * pointer2;</code>

Wertzuweisung an einen Pointer

1. Adressoperator

- Adressoperator **&** liefert Pointer auf Speicherplatz einer Variablen
- $\&Variable \rightsquigarrow$ Pointer auf *Datentyp* von *Variable*
- kann an Variable vom Typ Pointer auf *Datentyp* zugewiesen werden:
`pointer = α`

2. Wert einer anderen Pointervariablen

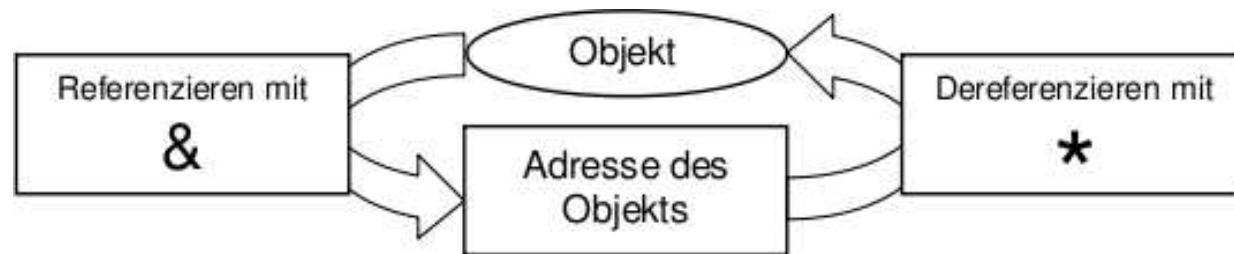
- `pointer2 = pointer1;`
- Übergabe der Adresse von einer Pointer-Variablen an eine weitere
- nur bei Pointern auf den gleichen Typ!

3. Konstante NULL

- vordefinierter Pointer, zeigt auf Adresse 0 / nie auf ein Speicherobjekt
- darf nie dereferenziert werden (\rightsquigarrow *Null-Pointer-Laufzeitfehler*)

Dereferenzieren

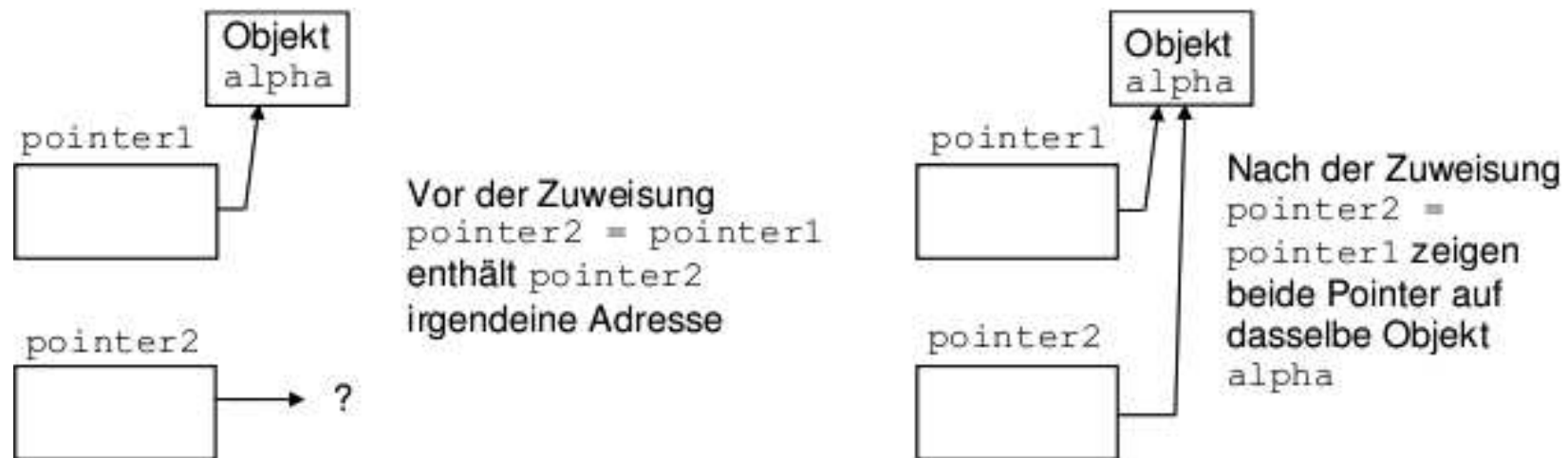
- **Inhaltsoperator** *: $*Pointer$
- * und & sind invers: $*\&\alpha$ ist äquivalent zu α



- $*pointer = Wert$; erlaubt
 \rightsquigarrow nicht bevor pointer gültige Adresse speichert!

Arbeit mit Pointern

- nicht initialisierte Variablen haben *irgendeinen* Wert!
- ⇒ nicht initialisierte Pointer verweisen auf *irgendeine* Adresse!
⇨ *Fehlerquelle!!!*



Rückblick: Benutzereingaben mit scanf

- formatiertes Einlesen eines Wertes mit scanf
- Formatelemente wie bei printf
- Beispiele: Einlesen einer ganzen Zahl und Speichern auf `int n`:
`scanf("%d", &n);`

Einlesen einer Gleitkommazahl und Speichern auf `double x`:
`scanf("%f", &x)`

Hinweis: Das Zeichen `&` ist nötig, da die Speicheradresse der Variablen angegeben werden muss (und kein Zugriff auf den Wert der Variablen erfolgt)

~> **Pointer können als Parameter an Funktionen übergeben werden!!!**

Parameterübergabe

Parameterübergabe mit call-by-value

- Vereinbarung einer lokalen Variablen:
Name und Typ des formalen Parameters
- *Kopie* des aktuellen Parameterwertes als Wert dieser lokalen Variablen
- Im Funktionsrumpf wird immer die lokale Variable verwendet.
 - ↪ **kein Einfluss auf aktuellen Parameter**
 - ↪ keine Seiteneffekte
- Ausdruck des aktuellen Parameters wird nur einmal ausgewertet
- verwendet in: C, C++, Java, C#, PASCAL, ...

call-by-value - Beispiel

```
int var1 = 1;  
int var2 = 2;
```

Aufgabe: Tausch der Werte von var1 und var2
mit einer Funktion swap(int m, int n)

```
void swap(int m, int n) {  
    int h = m;  
    m = n;  
    n = h;  
}
```

↪ Aufruf swap(var1, var2) ohne Effekt! (*Warum?*)

Andere Methoden der Parameterübergabe — nicht in C

- **call-by-value-result**

- zunächst wie call-by-value:
 - * lokale Variable mit Wert des aktuellen Parameters initialisiert
 - * keine Änderung am aktuellen Parameter
- am Ende (z.B. bei `return;`):
 - eine weitere automatische Wertzuweisung
 - ↗ Wert der lokalen Variablen → aktueller Parameter

- **call-by-name**

- textuelle Übergabe des aktuellen Bezeichners
- Im Funktionsrumpf wird jedes Vorkommen des formalen Parameters textuell ersetzt.
 - ↪ Seiteneffekt: kann aktuellen Parameter beeinflussen
- Ausdruck des aktuellen Parameters wird mehrfach ausgewertet
- kann zu Konflikten mit globalen Variablen führen
(falls Variablen des Funktionsrumpfes einen Bezeichner verwenden, der Teil eines aktuellen Parameters ist)

- **call-by-reference**

- Vereinbarung von lokalen Variablen für alle Variablen in den aktuellen Parametern
- referenzieren die Speicheradressen der Variablen in den aktuellen Parametern
- Zuweisungen beeinflussen die Werte der aktuellen Parameter direkt

- **call-by-reference**

- Vereinbarung von lokalen Variablen für alle Variablen in den aktuellen Parametern
- referenzieren die Speicheradressen der Variablen in den aktuellen Parametern
- Zuweisungen beeinflussen die Werte der aktuellen Parameter direkt
- **in C:**

Call-by-Reference-Semantik durch Übergabe von Pointern als Parameter

```
void swap(int *m, int *n) {  
    int h = *m;  
    *m = *n;  
    *n = h;  
}
```