



Wintersemester 2006/07

**Einführung in die Informatik für  
Naturwissenschaftler und Ingenieure**  
(alias **Einführung in die Programmierung**)  
(Vorlesung)

Prof. Dr. Günter Rudolph  
Fachbereich Informatik  
Lehrstuhl für Algorithm Engineering



## Kapitel 4: Zeiger



## Inhalt

- Zeiger
- Zeigerarithmetik

## Kapitel 4: Zeiger



## Caveat!

- Fehlermöglichkeiten immens groß!
- Falsch gesetzte Zeiger  $\Rightarrow$  Rechnerabstürze!

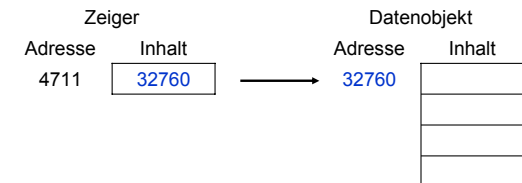
## Aber:

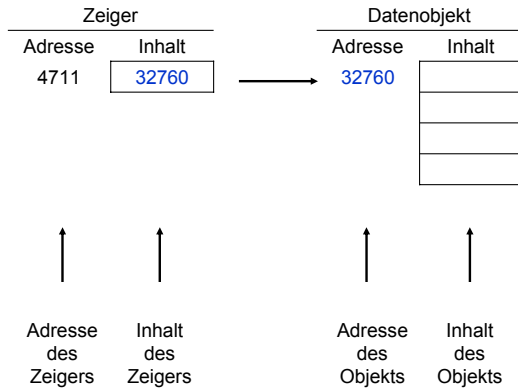
- Machtvolles Konzept!
- Deshalb genaues Verständnis unvermeidlich!
- Dazu müssen wir etwas ausholen ...

## Kapitel 4: Zeiger

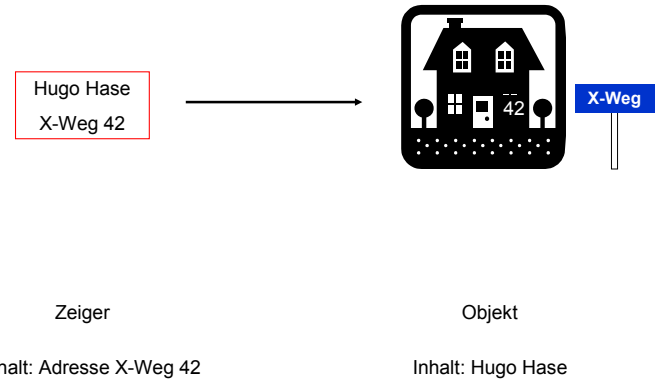


- Speicherplätzen sind fortlaufende Nummern (= Adressen) zugeordnet
- Datentyp legt Größe eines Datenobjektes fest
- Lage eines Datenobjektes im Speicher bestimmt durch Anfangsadresse
- **Zeiger** = Datenobjekt mit Inhalt (4 Byte)
- Inhalt interpretiert als Adresse eines **anderen** Datenobjektes





Beispiel: Visitenkarte



Zeiger: Wofür?

- Zeiger weiterreichen einfacher als Datenobjekt weiterreichen
- Zeiger verschieben einfacher / effizienter als Datenobjekt verschieben
- etc.

Datendefinition

Datentyp \*Bezeichner;

→ reserviert 4 Byte für einen Zeiger, der auf ein Datenobjekt vom Typ des angegebenen Datentyps verweist

Beispiel

- `double Umsatz;` ← „Herkömmliche“ Variable vom Type `double`
- `double *pUmsatz;` ← Zeiger auf Datentyp `double`

Was passiert genau?

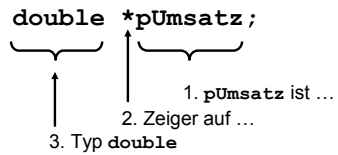
<code>double Umsatz;</code>	reserviert 8 Byte für Datentyp <code>double</code> ; symbolischer Name: <code>Umsatz</code> ; Rechner kennt jetzt Adresse des Datenobjektes
<code>double *pUmsatz;</code>	reserviert 4 Byte für einen Zeiger, der auf ein Datenobjekt vom Type <code>double</code> zeigen kann; symbolischer Name: <code>pUmsatz</code>
<code>Umsatz = 122542.12;</code>	Speicherung des Wertes <code>122542.12</code> an Speicherort mit symbolischer Adresse <code>Umsatz</code>
<code>pUmsatz = &amp;Umsatz;</code>	holt Adresse des Datenobjektes, das an symbolischer Adresse <code>Umsatz</code> gespeichert ist; speichert Adresse in <code>pUmsatz</code>
<code>*pUmsatz = 125000.;</code>	indirekte Wertzuweisung: Wert <code>125000.</code> wird als Inhalt an den Speicherort gelegt, auf den <code>pUmsatz</code> zeigt

**Zwei Operatoren: \* und &**

- \*-Operator:
  - mit Datentyp: Erzeugung eines Zeigers `double *pUmsatz;`
  - mit Variable: Inhalt des Ortes, an den Zeiger zeigt `*pUmsatz = 10.24;`
- &-Operator:
  - ermittelt Adresse des Datenobjektes `pUmsatz = &Umsatz;`

**Wie interpretiert man Datendefinition richtig?**

Man lese von rechts nach links!



**Initialisierung**

Sei bereits `double Umsatz;` vorhanden:

```
double *pUmsatz = &Umsatz;
```

oder

```
int *pINT = NULL;
```

oder

```
int *pINT = 0;
```

**Nullpointer!**  
Symbolisiert Adresse, auf der **niemals** ein Datenobjekt liegt!

**Verwendung Nullzeiger:** Zeiger zeigt auf Nichts! Er ist „leer“!

**Beispiele:**

```
double a = 4.0, b = 5.0, c;
c = a + b;
double *pa = &a, *pb = &b, *pc = &c;
*pc = *pa + *pb;
```

```
double x = 10.;
double y = *x;
```

**Typischer Fehler**

```
double *widerstand;
*widerstand = 120.5;
```

Dem Zeiger wurde keine Adresse zugewiesen!

Er zeigt also irgendwo hin:

- a) Falls in geschützten Speicher, dann **Abbruch** wg. Speicherverletzung! ☹
- b) Falls in nicht geschützten Speicher, dann Veränderung anderer Daten!  
**Folge:** Seltsames Programmverhalten! Schwer zu erkennender Fehler! ☹



Unterscheidung

• Konstante Zeiger

```
char *text = "Hallo";
int feld[] = { 2, 3, 4 };
```

zeigt auf feste Adresse im Speicher, auf die Programmierer nicht verändernd zugreifen kann!

Aber: Es gibt Compiler-spezifische Unterschiede!



```
int *const cpFeld = feld;
v.r.n.l.: cpFeld ist konstanter Zeiger auf Datentyp int
```

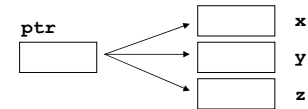


Unterscheidung

• Veränderliche Zeiger

```
double x = 2.0, y = 3.0, z = 7.0, s = 0.0, *ptr;
ptr = &x;
s += *ptr;
ptr = &y;
s += *ptr;
ptr = &z;
s += *ptr;
```

ptr nimmt nacheinander verschiedene Werte (Adressen) an  
s hat am Ende den Wert 12.0



Zeigerarithmetik

Sei  $T$  ein beliebiger Datentyp in der Datendefinition  $T *ptr$ ;  
und  $ptr$  ein Zeiger auf ein Feldelement eines Arrays von Typ  $T$

Dann bedeutet:

```
ptr = ptr + 1;      oder      ++ptr;
```

dass der Zeiger  $ptr$  auf das nächste Feldelement zeigt.

Analog:

```
ptr = ptr - 1;      oder      --ptr;
```

Zeiger  $ptr$  zeigt dann auf das vorherige Feldelement



Zeigerarithmetik

Achtung:

```
T val;
T *ptr = &val;
ptr = ptr + 2;
```

In der letzten Zeile werden **nicht** 2 Byte zu  $ptr$  hinzugezählt,  
sondern 2 mal die Speichergröße des Typs  $T$ .

Das wird auch dann durchgeführt wenn  $ptr$  nicht auf Array zeigt.

Zeigerarithmetik

```
int a[] = { 100, 110, 120, 130 }, *pa, sum = 0;
pa = &a[0];
sum += *pa + *(pa + 1) + *(pa + 2) + *(pa + 3);
```

```
struct KundeT {
    double umsatz;
    float skonto;
};
KundeT Kunde[5], *pKunde;
pKunde = &Kunde[0];
int i = 3;
*pKunde = *(pKunde + i);
```

Größe des Datentyps KundeT:  
8 + 4 = 12 Byte

Sei pKunde == 10000

Dann (pKunde + i) == 10036

Zeigerarithmetik

```
char *quelle = "Ich bin eine Zeichenkette";
char ziel[100], *pz;

// Länge der Zeichenkette
char *pq = quelle;
while (*pq != '\0') pq++;
int len = pq - quelle;

if (len < 100) {
    // Kopieren der Zeichenkette
    pq = quelle;
    pz = ziel;
    while (*pq != '\0') {
        *pz = *pq;
        pz++; pq++;
    }
}
*pz = '\0';
```

← Kommentar

← Kommentar

Das geht „kompakter!“  
später!

Zeiger auf Datenverbund (struct)

```
struct punktT { int x, y; };
punktT punkt[1000];
punktT *ptr = punkt;
punkt[0].x = 10;
punkt[2].x = 20;
punkt[k].x = 100;
```

↔

```
ptr->x = 10;
(ptr + 2)->x = 20;
(ptr + k)->x = 100;
```

(\*ptr).x ist identisch zu ptr->x