

UNIVERSITÄT DORTMUND

Wintersemester 2007/08

**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

Rudolph: EINI (WS 2007/08) • Kap. 4: Zeiger 2

Kapitel 4: Zeiger

Caveat!

- Fehlermöglichkeiten immens groß!
- Falsch gesetzte Zeiger => Programm- und zuweilen Rechnerabstürze!

Aber:

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

Rudolph: EINI (WS 2007/08) • Kap. 4: Zeiger 3

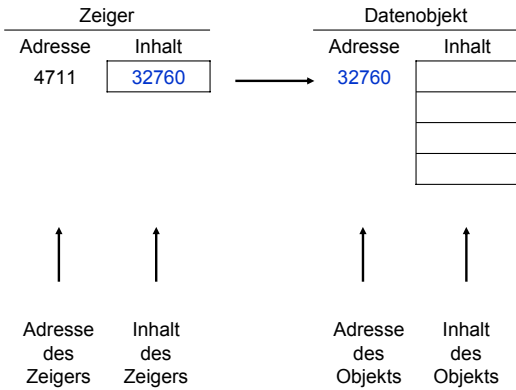
Kapitel 4: Zeiger

- 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

| | | | | |
|---------|--------|---|-------------|--------|
| Zeiger | | | Datenobjekt | |
| Adresse | Inhalt | | Adresse | Inhalt |
| 4711 | 32760 | → | 32760 | |
| | | | | |
| | | | | |
| | | | | |

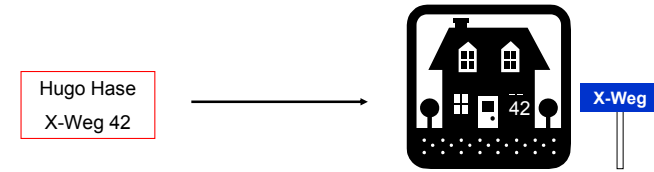
Rudolph: EINI (WS 2007/08) • Kap. 4: Zeiger 4

Kapitel 4: Zeiger



Kapitel 4: Zeiger

Beispiel: Visitenkarte



Zeiger

Objekt

Inhalt: Adresse X-Weg 42

Inhalt: Hugo Hase

Kapitel 4: Zeiger

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`

Kapitel 4: Zeiger

Was passiert genau?

`double Umsatz;`

reserviert 8 Byte für Datentyp `double`;
symbolischer Name: `Umsatz`;
Rechner kennt jetzt Adresse des Datenobjektes

`double *pUmsatz;`

reserviert 4 Byte für einen Zeiger,
der auf ein Datenobjekt vom Type `double` zeigen kann;
symbolischer Name: `pUmsatz`

`Umsatz = 122542.12;`

Speicherung des Wertes `122542.12` an Speicherort
mit symbolischer Adresse `Umsatz`

`pUmsatz = &Umsatz;`

holt Adresse des Datenobjektes,
das an symbolischer Adresse `Umsatz` gespeichert ist;
speichert Adresse in `pUmsatz`

`*pUmsatz = 125000.;`

indirekte Wertzuweisung:
Wert `125000.` wird als Inhalt an den Speicherort gelegt,
auf den `pUmsatz` zeigt

Kapitel 4: Zeiger

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!

```
double *pUmsatz;
```

1. pUmsatz ist ...
2. Zeiger auf ...
3. Typ double

Kapitel 4: Zeiger

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“!

Kapitel 4: Zeiger

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;
```

Kapitel 4: Zeiger

Typischer Fehler

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

Dem Zeiger wurde **keine Adresse** zugewiesen!

Er zeigt also irgendwo hin:

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

Kapitel 4: Zeiger

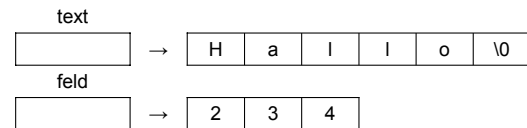
Unterscheidung

- **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

Exkurs: const Qualifizierer

Schlüsselwort `const` gibt an, dass Werte nicht verändert werden können!

Leider gibt es 2 Schreibweisen:

```
const int a = 1;      identisch zu      int const a = 1;
```

⇒ konstanter Integer

⇒ da Konstanten kein Wert zuweisbar, Wertbelegung bei Initialisierung!

verschiedene Schreibweisen können zu Verwirrungen führen ...

(besonders bei **Zeigern** !)

⇒ am besten konsistent bei einer Schreibweise bleiben!

Exkurs: const Qualifizierer

Fragen:

1. Was ist konstant?
2. Wo kommt das Schlüsselwort `const` hin?

```
char* s1               = "Zeiger auf char";  
char const* s2        = "Zeiger auf konstante char";  
char* const s3        = "Konstanter Zeiger auf char";  
char const* const s4 = "Konstanter Zeiger auf konstante char";
```

Sinnvolle Konvention / Schreibweise:

Konstant ist, was **vor** dem Schlüsselwort `const` steht!

⇒ Interpretation der Datendefinition / Initialisierung von rechts nach links!

Exkurs: const Qualifizierer

| | | Zeiger | Inhalt | |
|-----------------------------------|---------------------|--------|--------|-----------------|
| <code>char* s1</code> | <code>= "1";</code> | V | V | |
| <code>char const* s2</code> | <code>= "2";</code> | V | K | V: veränderlich |
| <code>char* const s3</code> | <code>= "3";</code> | K | V | K: konstant |
| <code>char const* const s4</code> | <code>= "4";</code> | K | K | |

```
*s1 = 'a';  
*s2 = 'b';   // Fehler: Inhalt nicht veränderbar  
*s3 = 'c';  
*s4 = 'd';   // Fehler: Inhalt nicht veränderbar
```

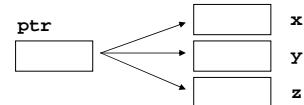
```
char* s0 = "0";  
s1 = s0;  
s2 = s0;  
s3 = s0;    // Fehler: Zeiger nicht veränderbar  
s4 = s0;    // Fehler: Zeiger nicht veränderbar
```

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