technische universität dortmund

Einführung in die Programmierung

Wintersemester 2011/12

Prof. Dr. Günter Rudolph
Lehrstuhl für Algorithm Engineering
Fakultät für Informatik
TU Dortmund

Inhalt

- Zeiger
- Zeigerarithmetik
- Zeiger für dynamischen Speicher
- Anwendungen

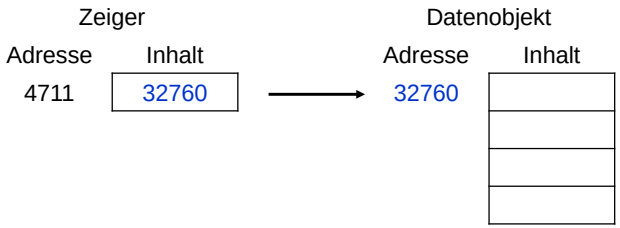
Caveat!

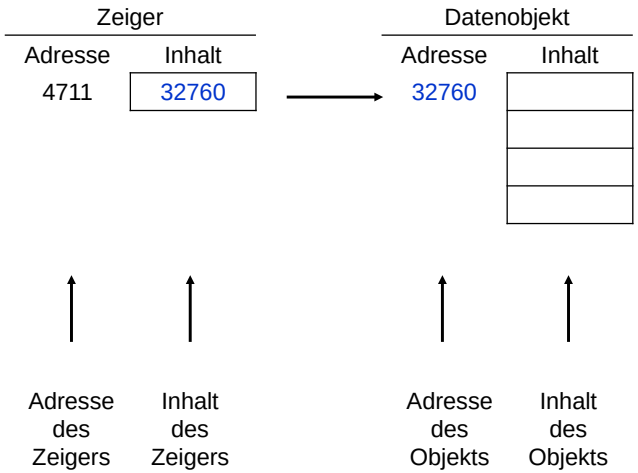
- Fehlermöglichkeiten immens groß!
- Falsch gesetzte Zeiger $\Rightarrow$ Programm- und zuweilen Rechnerabstürze!

Aber:

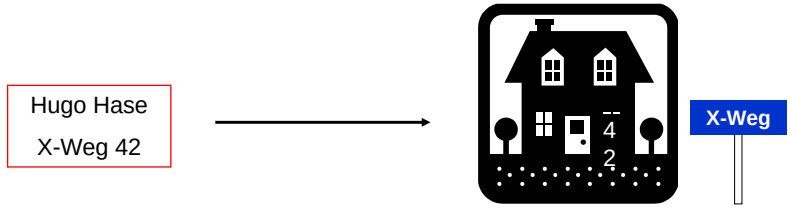
- Machtvolles Konzept!
- Deshalb genaues Verständnis unvermeidlich!
- Dazu müssen wir etwas ausholen ...

- 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

Inhalt: Adresse X-Weg 42

Objekt

Inhalt: Hugo Hase

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?

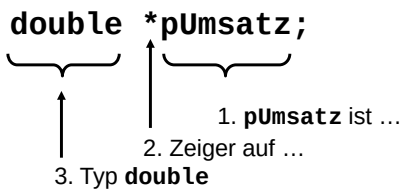
`double Umsatz;`	reserviert 8 Byte für Datentyp **`double`**; symbolischer Name: **`Umsatz`**; Rechner kennt jetzt Adresse des Datenobjektes
`double *pUmsatz;`	reserviert 4 Byte für einen Zeiger, der auf ein Datenobjekt vom Type **`double`** zeigen kann; symbolischer Name: **`pUmsatz`**
`Umsatz = 122542.12;`	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

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! ☹
  ```

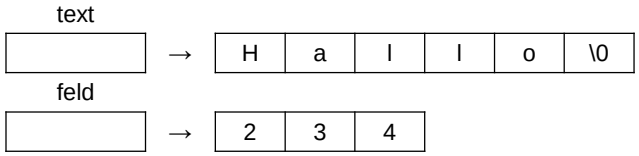
Unterscheidung

- **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 **const**anter Zeiger auf Datentyp **int**

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!

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!

	Zeiger	Inhalt
`char* s1 = "1";`	V	V
`char const* s2 = "2";`	V	K
`char* const s3 = "3";`	K	V
`char const* const s4 = "4";`	K	K

V: veränderlich
K: konstant

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

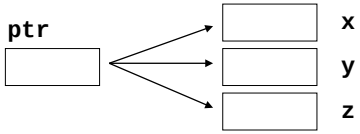
```
char* s0 = "0";
s1 = s0;
s2 = s0;
s3 = s0;      // Fehler: Zeiger nicht veränderbar
s4 = s0;      // Fehler: Zeiger nicht veränderbar
```

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";
int const maxZeichen = 100;
char ziel[maxZeichen] , *pz = &ziel[0];

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

if (len < maxZeichen) {
  // Kopieren der Zeichenkette
  pq = quelle;
  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`

Aufgabe:

Lese zwei Vektoren reeller Zahlen der Länge n ein.

$a = (a_1, \ldots, a_n)'$ $b = (b_1, \ldots, b_n)'$

Berechne das Skalarprodukt ... $\sum_{i=1}^{n} a_i \cdot b_i$

… und gebe den Wert auf dem Bildschirm aus!

Lösungsansatz:

Vektoren als Arrays von Typ **`double`**.

n darf höchstens gleich der Arraygröße sein! Testen und ggf. erneute Eingabe!


```
#include <iostream>
using namespace std;

int main() {
  unsigned int const nmax = 100;
  unsigned int i, n;
  double a[nmax], b[nmax];

  // Dimension n einlesen und überprüfen
  do {
    cout << "Dimension ( n < " << nmax << " ): ";
    cin >> n;
  } while (n < 1 || n > nmax);
```

(Fortsetzung folgt …)

Datendefinition **`double a[nmax]`** OK, weil **`nmax`** eine Konstante ist!

Ohne **`const`**: Fehlermeldung!
z.B. „Konstanter Ausdruck erwartet"

Der aktuelle GNU C++ Compiler erlaubt folgendes:

```
#include <iostream>
int main() {
  int n;
  std::cin >> n;
  double a[n];
  a[0] = 3.14;
  return 0;
}
```

Aber: Der Microsoft C++ Compiler (VS 2003) meldet einen Fehler!

Variable Arraygrenzen sind nicht Bestandteil des C++ Standards!

Verwendung von Compiler-spezifischen Spracherweiterungen führt zu Code, der **nicht portabel** ist! ☹ 💣 ☠

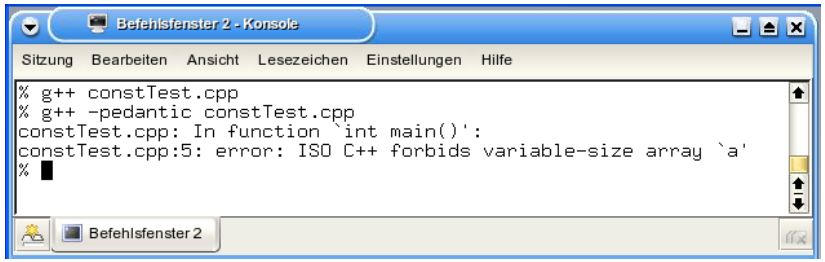
Das ist nicht wünschenswert!


```
#include <iostream>
int main() {
  int n;
  std::cin >> n;
  double a[n];
  a[0] = 3.14;
  return 0;
}
```

Also: Bei Softwareentwicklung nur Sprachelemente des C++ Standards verwenden!

Bei GNU Compiler: Option **`-pedantic`**

C++ Standard ISO/IEC 14882-2003
z.B. als PDF-Datei erhältlich für 30$
`http://webstore.ansi.org/`



(… Fortsetzung)

```
  // Vektor a einlesen
  for (i = 0; i < n; i++) {
    cout << "a[" << i << "]= ";
    cin >> a[i];
  }
  // Vektor b einlesen
  for (i = 0; i < n; i++) {
    cout << "b[" << i << "]= ";
    cin >> b[i];
  }
  // Skalarprodukt berechnen
  double sp = 0.;
  for (i = 0; i < n; i++)
    sp += a[i] * b[i];
  // Ausgabe
  cout << "Skalarprodukt = " << sp << endl;

  return 0;
}
```

Anmerkung:

Fast identischer Code!

Effizienter mit Funktionen!

→ nächstes Kapitel!

Unbefriedigend bei der Implementierung:

Maximale festgelegte Größe des Vektors!

→ Liegt an der unterliegenden Datenstruktur Array:

Array ist **statisch**, d.h.
Größe wird zur Übersetzungszeit **festgelegt**
und ist **während** der **Laufzeit** des Programms **nicht veränderbar**!

Schön wären **dynamische** Datenstrukturen, d.h.
Größe wird zur Übersetzungszeit **nicht festgelegt**
und ist **während** der **Laufzeit** des Programms **veränderbar**!

Das geht mit **dynamischem Speicher** …

… und **Zeigern**!

Erzeugen und Löschen eines Objekts zur Laufzeit:

1. Operator **new** erzeugt Objekt
2. Operator **delete** löscht zuvor erzeugtes Objekt

Beispiel: (Erzeugen)

```
int *xp    = new int;
double *yp = new double;

struct PunktT {
  int x, y;
};

PunktT *pp = new PunktT;
```

Beispiel: (Löschen)

```
delete xp;
delete yp;



delete pp;
```

```
int n       = 10;
int *xap    = new int[n];
PunktT *pap = new PunktT[n];
```

```
delete[] xap;
delete[] pap;
```

variabel, nicht konstant!

Bauplan:

Datentyp *Variable = **new** Datentyp; (Erzeugen)

delete Variable; (Löschen)

Bauplan für Arrays:

Datentyp *Variable = **new** Datentyp[Anzahl]; (Erzeugen)

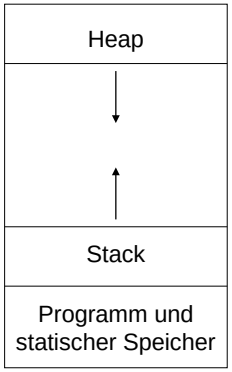
delete[] Variable; (Löschen)

Achtung:

Dynamisch erzeugte Objekte müssen auch wieder gelöscht werden!
Keine automatische Speicherbereinigung!

Wo wird Speicher angelegt?

⇒ im **Freispeicher** alias **Heap** alias **dynamischen Speicher**



wächst nach „unten“

wächst nach „oben“

wenn Heapgrenze auf Stackgrenze stösst:

Out of Memory Error

⇓

Stack bereinigt sich selbst, für Heap ist Programmierer verantwortlich!

Zurück zur **Beispielaufgabe:**

```
unsigned int const nmax = 100;
unsigned int i, n;
double a[nmax], b[nmax];

do {
  cout << "Dimension ( n < " << nmax << " ): ";
  cin >> n;
} while (n < 1 || n > nmax);
```

vorher:

statischer Speicher

```
unsigned int i, n;
double *a, *b;

do {
  cout << "Dimension: ";
  cin >> n;
} while (n < 1);
a = new double[n];
b = new double[n];
```

nachher:

dynamischer Speicher

Nicht vergessen:

Am Ende angeforderten dynamische Speicher wieder freigeben!

```
  delete[] a;
  delete[] b;

  return 0;
}
```

Sonst „Speicherleck“!

⇒ Programm terminiert möglicherweise anormal mit Fehlermeldung!

Beispiel für **programmierten Absturz**:

```
#include <iostream>
using namespace std;

int main() {
  unsigned int const size = 100 * 1024;
  unsigned short k = 0;

  while (++k < 5000) {
    double* ptr = new double[size];
    cout << k << endl;
    // delete[] ptr;
  }
  return 0;
}
```

bei **k** ≈ 2500 sind 2 GB erreicht

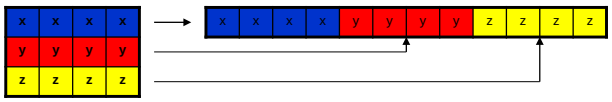
⇒ Abbruch!

Projekt: Matrix mit dynamischem Speicher (Größe zur Laufzeit festgelegt!)

Vorüberlegungen:

Speicher im Rechner ist **linear**!

⇒ Rechteckige / flächige Struktur der Matrix linearisieren!



n Zeilen, m Spalten ⇒ n x m Speicherplätze!

Projekt: Matrix mit dynamischem Speicher (Größe zur Laufzeit festgelegt!)

```
double **matrix;

matrix = new double*[zeilen];

matrix[0] = new double[zeilen * spalten];

for (i = 1; i < zeilen; i++)
  matrix[i] = matrix[i-1] + spalten;
```

Zugriff wie beim zweidimensionalen statischen Array:

```
matrix[2][3] = 2.3;
```