C++ Teil 5

Sven Groß

16. Nov 2015

Themen der letzten Vorlesung

- Namensräume
- Live Programming zu A2
- Gleitkommazahlen
  - Rundungsfehler
  - Auswirkung auf Vergleiche
- Funktionen
  - Überladung, Signatur

Heutige Themen

1. Typumwandlung (Cast)
2. Funktionen
   - Rekursion
3. const-Deklaration
4. Referenzen
5. Zeiger
6. Felder

Impliziter und expliziter Cast

- automatische, **implizite Typumwandlung** (**Cast**), leider auch bei Informationsverlust,
  z.B. `int` $\rightleftarrows$ `double`, `float` $\rightleftarrows$ `double`, `int` $\rightleftarrows$ `bool`

$\rightsquigarrow$ oft nützlich, aber manchmal auch gefährlich.

- Typumwandlung kann auch **explizit** angefordert werden:

```
int a=2, b=5;

double c= a / b,                        // 0, int-Division!

       d= double(a) / double(b), // 0.4, C-style cast

       e= static_cast<double>( b ) / a;
            // 2.5, C++-style cast + impliziter Cast
```

- Typumwandlung von/nach `string` funktioniert (leider) anders, dazu später mehr.

Rekursive Funktionen

- **Rekursive Funktionen** rufen sich selber auf
- sicherstellen, dass Rekursion abbricht!
- einfache Rekursion kann auch durch Iteration (Schleife) ersetzt werden

```
int fakultaet_rekursiv( int n)
{
    if (n>1)
        return n*fakultaet_rekursiv(n-1);
    else
        return 1;     // Abbruch der Rekursion
}

int fakultaet_iterativ( int n)
{
    int fak= 1;
    for (int i=1; i<=n; ++i)
        fak*= i;

    return fak;
}
```

Rekursive Funktionen (2)

Beispiel: Fibonacci-Zahlen: $a_0 := a_1 := 1, \quad a_n := a_{n-2} + a_{n-1}$ für $n \geq 2$

```
int fib_rekursiv( int n)
{
    if (n >= 2)
      return fib_rekursiv( n-2) + fib_rekursiv( n-1);
    else
      return 1;     // Abbruch der Rekursion
}
```

- Rekursion hier sehr ineffizient!
  - `fib_rekursiv( n )` erzeugt $\mathcal{O}(2^{n-1})$ Aufrufe von sich selbst.
  - Bei Berechnung von a_n wird z.B. a_2 sehr oft neu berechnet.

$\rightsquigarrow$ **besser:** Rekursion durch Iteration ersetzen

Rekursive Funktionen (3)

Beispiel: Fibonacci-Zahlen: $a_0 := a_1 := 1, \quad a_n := a_{n-2} + a_{n-1}$ für $n \geq 2$

```
int fib_iterativ( int n)
{
    if (n < 2)
        return 1;

    int ak, ak1= 1, ak2= 1;  // a_k, a_{k-1}, a_{k-2}

    for (int k= 2; k <= n; ++k)
    {
        ak= ak2 + ak1;       // nach Definition
        ak2= ak1;  ak1= ak; // Variablen shiften
    }
    return ak;
}
```

Iterativer Ansatz ist deutlich effizienter:

- Aufwand zur Berechnung von a_n ist linear in n (nicht exponentiell)
- a_2 wird genau einmal berechnet (für `k=2`)

const - Deklaration

- Das Schlüsselwort `const` vor dem Datentyp erzeugt **konstante** Variablen.

```
const double Pi = 3.1415;
```

- Konstante Variablen dürfen gelesen,
  aber **nicht** verändert werden.
- taucht oft im Zusammenhang mit Funktionsparametern auf:

```
void write_perReference( const RiesigerDatenTyp& v);
```

  - Referenzparameter vermeidet Kopieren des Arguments (sinnvoll bei großen Datentypen),
  - `const`-Deklaration schützt vor unabsichtlichem Verändern.

Referenzen

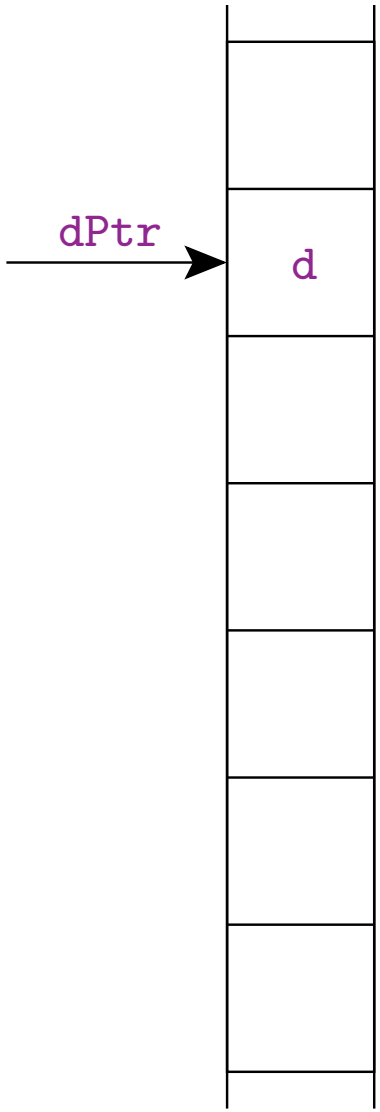
- bereits bekannt im Zusammenhang mit Referenzparametern, können aber auch woanders verwendet werden
- Referenz dient als **Alias** für bestehende Variable, die dann unter neuem Namen angesprochen werden kann
- Referenzen *müssen* mit Variable initialisiert werden, auf die sie verweisen

```
int a= 3;
int &r= a;
// a hat Wert 3, r verweist auf 3

a+= 4;
// a hat Wert 7, r verweist auf 7

r-= 2;
// a hat Wert 5, r verweist auf 5
```

Zeiger (*Pointer*)

- Variable, die eine **Speicheradresse** enthält

```
int    i= 5;
double d= 47.11;

int    *iPtr; // Zeiger auf int
double *dPtr; // Zeiger auf double
```



- **Adressoperator** `&` liefert Zeiger auf Variable
  (nicht zu verwecheln mit Referenz-Deklaration)

```
iPtr= &i;
dPtr= &d;
```

- **Dereferenzierungsoperator** `*` liefert Wert an Speicheradresse
  (nicht zu verwecheln mit Zeiger-Deklaration)

```
cout << (*dPtr);   // gibt d aus
(*iPtr)= 42;       // i ist jetzt 42
```

Zeiger (2)

- synonym: `int *iPtr;` oder `int* iPtr;` oder `int * iPtr;`

```
int* iPtr, j, *kPtr;   // zwei int-Zeiger und ein int (j)

double * d_zeiger,     // ein double-Zeiger
       *e_zeiger;      // noch ein double-Zeiger
```

- Speicheradressen werden meist im Hexadezimalsystem (0,...,9,A,...,F) angegeben, `cout << iPtr;` liefert z.B. Ausgabe `0xBF` $= 11 \cdot 16 + 15 = 191$
- **Nullzeiger** zeigt ins „Nirgendwo“, signalisiert leeren Zeiger:

```
iPtr= 0;
```

  Initialisierung mit `0` oder `NULL` (in C++11: `nullptr`)

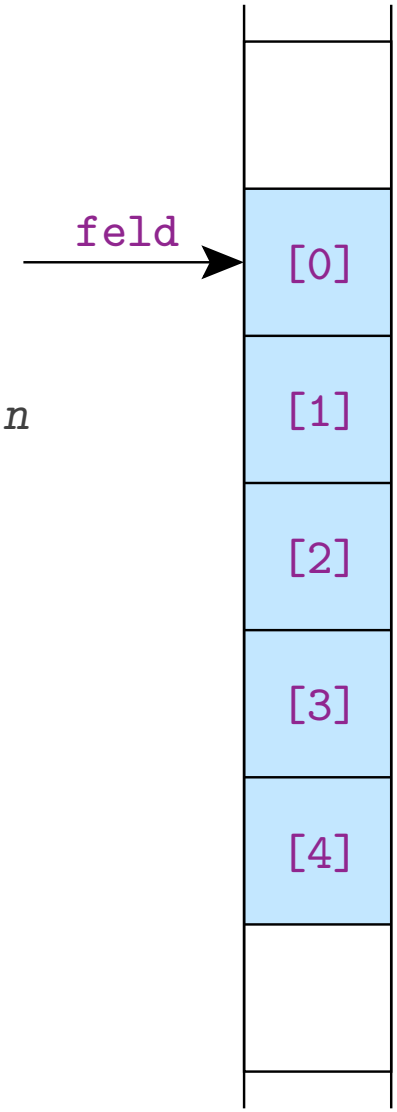
- `const`-Qualifizierung für Zeiger und/oder Datentyp möglich:

```
const int * p1;   // Zeiger auf konstanten int
int * const p2;   // konstanter Zeiger auf int
```

  $\rightsquigarrow$ Eselsbrücke: von rechts nach links lesen!

Felder (*Arrays*)

- Wichtiger Verwendungszweck für Zeiger: **Felder**
- ⚠ Nummerierung der Einträge beginnt bei 0, endet bei n-1
- Zugriff auf Einträge mit `[ ]`-Operator
- keine Bereichsprüfung (Speicherverletzung !!)




```
double feld[5];   // legt ein Feld von 5 doubles an
                  // (nicht initialisiert),
                  // feld ist ein double-Zeiger

for (int i=0; i<5; ++i)
  feld[i]= 0.7;   // setzt Eintraege 0...4

cout << (*feld); // gibt ersten Eintrag aus
                 // (feld[0] und *feld synonym)

cout << feld[5]; // ungueltiger Zugriff!
                 // Kein Compilerfehler, evtl.
                 // Laufzeitfehler (segmentation fault)
```

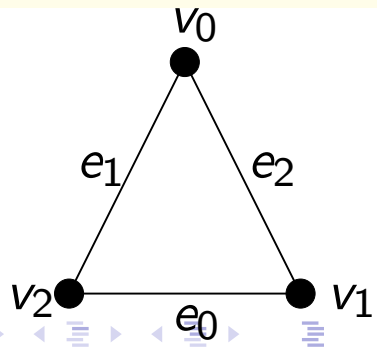
Initialisierung von Feldern

Felder können bei Erzeugung auch **initialisiert** werden,
in diesem Fall muss die Länge nicht spezifiziert werden.

```
double werte[5]= { 1.1, 2.2, 3.3, 4.4, 5.5};
    // legt ein Feld von 5 doubles an
    // und initialisiert es mit geg. Werten
double zahl[5]=  { 1.1, 2.2}; // (Rest mit Null initial.)

int lottozahlen[]= { 1, 11, 23, 29, 36, 42};
    // legt ein Feld von 6 ints an (ohne Laengenangabe)
    // und initialisiert es mit geg. Werten

int vertex_of_edge[3][2]= { {1, 2}, {0, 2}, {0, 1} };
    // legt ein zweidimensionales int-Feld an
    // mit 3x2 gegebenen Werten. Typ: int**
```

Quiz:

- Welchen Typ hat der Ausdruck `vertex_of_edge[1]`?
- Wie kann ich den 1. Eckpunkt der 3. Kante abfragen?



Kopieren von Feldern

```
double werte[5]= { 1.1, 2.2, 3.3, 4.4, 5.5};

double *nochEinFeld= werte;  // kopiert nur den Zeiger !!!
nochEinFeld[2]= 99;          // veraendert werte[2] !!!

double auchFalsch[5]; // reserviert neuen Speicher (5 doubles)
// auchFalsch= werte; // Compilerfehler

double richtig[5];  // neuen Speicher reservieren

for (int i=0; i<5; ++i)
    richtig[i]= werte[i]; // Werte kopieren
```

⚠ Vorsicht beim Kopieren von Feldern!

- Zuweisung = verändert nur den Zeiger, nicht aber den Feldinhalt
- Kopie benötigt eigenen Speicher
- Eintrag für Eintrag kopieren
- Alternative für Felder: **Vektoren** bequem mit = kopieren (dazu später mehr)

Spezielle char-Felder: C-Strings

Relikt aus C-Zeiten: **C-Strings**

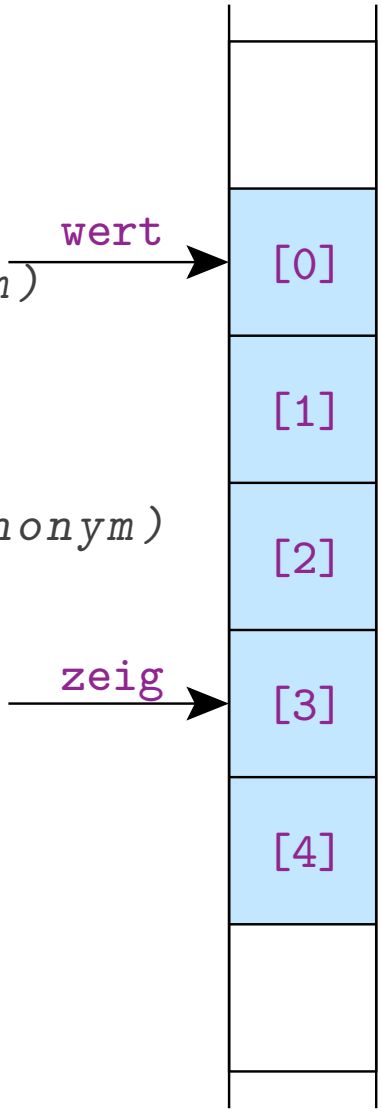
- char-Felder
- hören mit Terminationszeichen '\0' auf
- Initialisierung mit "..."

```
char cstring[]= "Hallo";
    // Zeichenkette mit 5 Zeichen, aber Feld mit 6 chars
    // { 'H', 'a', 'l', 'l', 'o', '\0' }

char message[]= "Hier␣spricht␣Edgar␣Wallace\n";
    // noch ein C-String

string botschaft= message;
    // C++-String, der mit C-String initialisiert wird

const char *text= botschaft.c_str();
    // ...und wieder als C-String
```

Zeigerarithmetik

- Zeiger + `int` liefert Zeiger:

```
double wert[5]= { 1.1, 2.2, 3.3, 4.4, 5.5};

cout << (*wert); // gibt ersten Eintrag aus
                 // (wert[0] und *wert synonym)

double *zeig= wert + 3;
cout << (*zeig); // gibt vierten Eintrag aus
                 // (wert[3] und *(wert+3) synonym)

double *ptr= &(wert[2]);
ptr++;
bool same= (zeig == ptr);  // true
```



- Zeiger – Zeiger liefert `int` (Abstand):

```
int differenz= zeig - ptr,
    index= zeig - wert;
```

Quiz: Werte von `differenz` und `index`?