C++ Teil 6

Sven Groß

23. Nov 2015

Themen der letzten Vorlesung

- const-Deklaration
- Referenzen
- Zeiger
- Felder

Heutige Themen

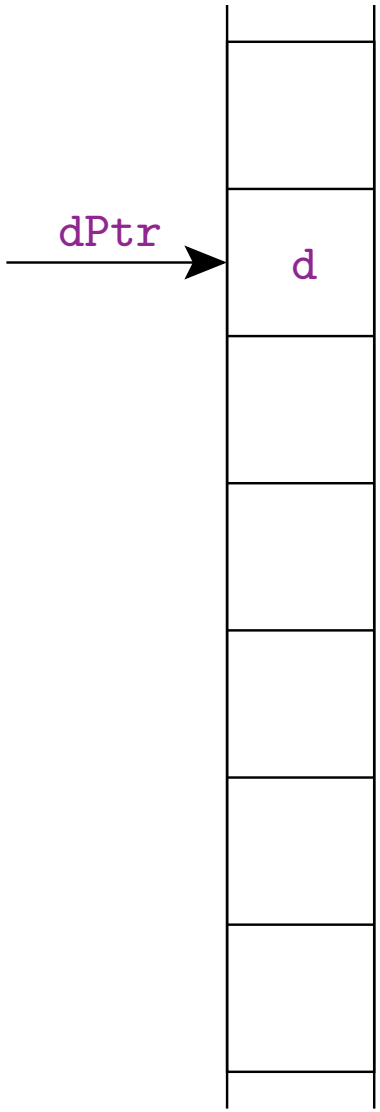
1. Wdh.: Zeiger und Felder
2. Dynamische Speicherverwaltung
3. Casts bei Zeigern

Wdh.: 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
```

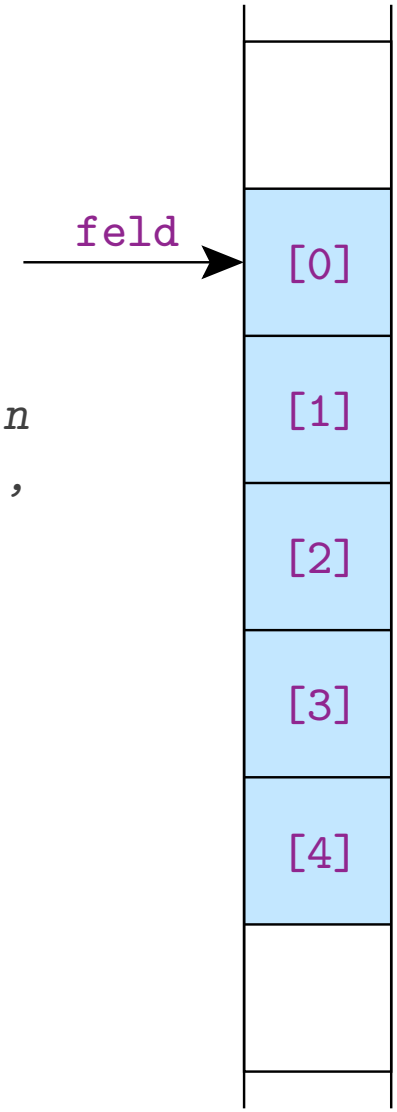
Beispiel: Referenzen und Zeiger

```
1  int a= 3;
2  int &r= a;  // int-Referenz auf a
3  int *p= &a; // int-Zeiger auf a
4  // a hat Wert 3,  r verweist auf 3,  p zeigt auf 3
5
6  a+= 4;
7  cout << (*p) << endl;
8  // a hat Wert 7,  r verweist auf 7,  p zeigt auf 7
9
10 r-= 2;
11 cout << a << endl;
12 // a hat Wert 5,  r verweist auf 5,  p zeigt auf 5
13
14 (*p)= r*a;
15 cout << r << endl;
16 // a hat Wert ?,  r verweist auf ?,  p zeigt auf ?
```

Quiz: Was verändert sich, wenn wir `p` als Zeiger auf `const int` deklarieren? Wie würde die Deklaration von `p` aussehen?

Wdh.: 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
    { 1, 2};      // (initialisiert mit 1 2 0 0 0),
                  // feld ist ein double-Zeiger

for (int i=2; i<5; ++i)
  feld[i]= 0.7;   // setzt Eintraege 2...4

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

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

Wdh.: 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 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

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

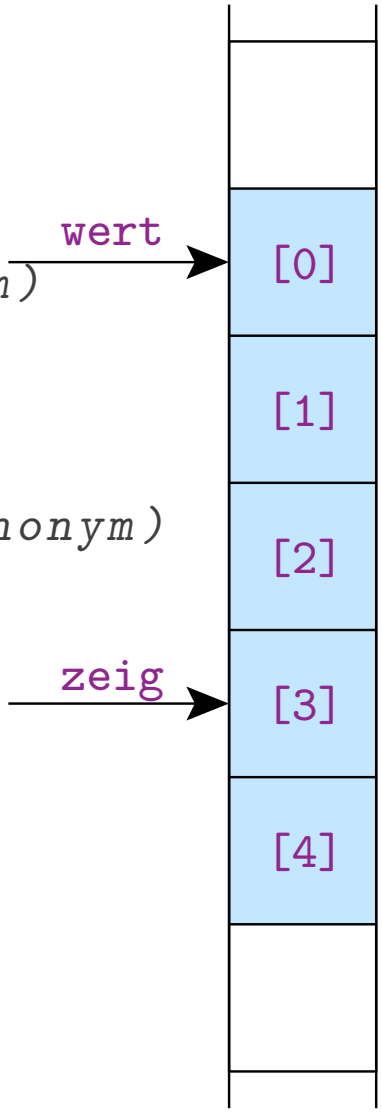
- Zeiger + `int` liefert Zeiger:

```
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`?

Beispiel: so geht es nicht...

Beispiel: Funktion, die ein Feld kopieren soll

```
1  double* copy( double* original, int n)
2  {
3      double kopie[n];  // statisches Feld
4      for (int i=0; i<n; ++i)
5          kopie[i]= original[i];
6      return kopie;
7  }
8
9  int main()
10 {
11     double zahlen[3]= { 1.2, 3.4, 5.6};
12     double *zahlenkopie;
13     zahlenkopie= copy( zahlen, 3);
14     ...
15     return 0;
16 }
```

⚠ Das funktioniert so nicht!
Quiz: **Wieso?**

Dynamische Speicherverwaltung

- **dynamische Felder**,
  - falls Speicher vor Verlassen des Scopes freigegeben werden soll
  - falls Speicher nach Verlassen des Scopes freigegeben werden soll
  - (in C: falls Länge erst zur Laufzeit festgelegt werden soll)
- Speicherplatz belegen (*allocate*) mit `new`
- Speicherplatz freigeben (*deallocate*) mit `delete`,
  passiert **nicht automatisch**!

```
int *feld= new int[5]; // legt dyn. int-Feld der Laenge 5 an
int *iPtr= new int;    // legt dynamisch neuen int an
...
delete[] feld;         // Speicher wieder freigeben
delete   iPtr;
```

Strikte Regel: Auf jedes `new` (bzw. `new[]`) muss später ein zugehöriges `delete` (bzw. `delete[]`) folgen!

Dynamische Speicherverwaltung (2)

- **normale Variablen:** (automatische Speicherverwaltung)
  - werden erzeugt bei Variablendeklaration
  - werden automatisch zerstört bei Verlassen des Scope
- **dynamische Variablen:** (dynamische Speicherverwaltung)
  - werden erzeugt mit `new` (liefert Zeiger auf neuen Speicher zurück)
  - werden zerstört mit `delete` (Freigabe des Speichers)

  $\rightsquigarrow$ volle Kontrolle = volle Verantwortung!

Dynamische Speicherverwaltung (3)

```
1  {
2      double x= 0.25, *dPtr= 0; // neue double-Var. x,
3                                // neuer double-Zeiger dPtr
4
5      int *iPtr= 0, a= 33;      // neue int-Var. a,
6                                // neuer int-Zeiger iPtr
7      {
8          double y= 0.125;   // neue double-Var. y
9          dPtr= new double;  // neue double-Var. (ohne Namen)
10         *dPtr= 0.33;
11         iPtr= new int[4];  // neues int-Feld (ohne Namen)
12
13     } // Ende des Scope: y wird automatisch zerstoert
14     for (int i=0; i<4; ++i)
15         iPtr[i]= i*i;
16     ...
17     delete[] iPtr; // int-Feld wird zerstoert
18     delete dPtr;   // double-Var. wird zerstoert
19
20 } // Ende des Scope: Var. x, a werden automatisch zerstoert,
21   //          Zeiger dPtr, iPtr werden automatisch zerstoert.
```

Beispiel: so ist es richtig

Beispiel: Funktion, die ein Feld kopieren soll

```
1  double* copy( double* original, int n)
2  {
3      double *kopie= new int[n];  // dynamisches Feld
4      for (int i=0; i<n; ++i)
5          kopie[i]= original[i];
6      return kopie;
7  }
8
9  int main()
10 {
11     double zahlen[3]= { 1.2, 3.4, 5.6};
12     double *zahlenkopie;
13     zahlenkopie= copy( zahlen, 3);
14     ...
15     delete[] zahlenkopie;  // Speicher wieder frei geben
16     return 0;
17 }
```

$\rightsquigarrow$ `zahlenkopie` zeigt auf gültigen Speicherbereich, da das mit `new int[5]` erzeugte dynamische Feld erst am Ende von `main()` mit `delete[]` zerstört wird.

Casts bei Zeigern

```
void Ausgabe( const double* feld, int n)
{
    ...
}

int main()
{
    double data[]= { 47, 11, 0, 8, 15};

    Ausgabe( data, 5);
    return 0;
}
```

- Impliziter Cast `double*` → `const double*` bei Aufruf von `Ausgabe`
- `const` ermöglicht nur Lesezugriff auf Einträge von `feld`, verhindert, dass Einträge von `data` versehentlich in der Funktion `Ausgabe` verändert werden
- **Best Practice:** `const` benutzen, wenn möglich ⇝ mehr Datensicherheit