C++ Teil 7

Sven Groß

30. Nov 2015

Themen der letzten Vorlesung

- Zeiger, Felder (Wdh.)
- dynamische Speicherverwaltung

Heutige Themen

1. Casts bei Zeigern
2. Wdh.: Dynamische Speicherverwaltung
3. Vektoren
4. Typedefs und Makros

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

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.

Vektoren

- C++-Alternative zu C-Feldern (*Arrays*)
- ein Datentyp aus der STL (*Standard Template Library*)
- Datentyp der Einträge bei Deklaration angeben, z.B. `vector<double>`
- Angabe der Größe bei Deklaration oder später mit `resize`,
  Abfrage der Größe mit `size`
- zusammenhängender Speicherbereich garantiert,
  aber möglicherweise keine feste Adresse
- Zugriff auf Einträge mit `[ ]`-Operator:
  Nummerierung 0, ..., n-1, keine Bereichsprüfung!

```
#include <vector>
using std::vector;

vector<double> v(2), w;
w.resize(5);

for (int i=0; i < w.size(); ++i)
    w[i]= 0.07;   // alternativ:  w.at(i)= 0.07;
```

Vektoren (2)

```
w.push_back( 4.1); // haengt neuen Eintrag hinten an

v.pop_back();      // letzten Eintrag loeschen

double* ptr= &(v[0]);

v= w;              // Zuweisung, passt Groesse von v an

// ptr zeigt evtl auf ungueltigen Speicherbereich!!
```

- Zuweisung `=` funktioniert wie erwünscht (im Gegensatz zu Feldern)
- kein Vektor im mathematischen Sinne:
  arithmetische Operatoren (`+, -, *`) nicht implementiert
- Ändern der Länge führt evtl. zu Reallozierung an anderer Speicheradresse
  $\rightsquigarrow$ Vorsicht mit Zeigern!
- STL definiert weitere **Container** wie `list, map`, dazu später mehr
- Container verwenden **Iteratoren** statt Zeiger, auch dazu später mehr

Datentyp-Alias mit typedef

Neuer Name (Alias) für Datentypen mittels `typedef`, z.B.

- um Datentyp nachträglich einfach ändern zu können (nur 1 `typedef` ändern statt 100 Variablendeklarationen im Code),
- um Schreibarbeit zu sparen.

```
typedef double RealT;
// typedef float Real;

RealT x= 3.14;    // double-Var. x   (bzw. float-Var.)

typedef double* dZeigerT; // Datentyp fuer double-Zeiger

dZeigerT zeig, // double-Zeiger
    zfeld[4];  // Feld von double-Zeigern

typedef vector<int>         iVectorT;
typedef iVectorT::iterator  iVecIterT;

iVectorT ivec(5);
iVecIterT it= ivec.begin();
```

Präprozessor-Makros mit #define

C-Style, aber manchmal nützlich: **Präprozessor-Makros**

```
#define REAL double
#define SIZE 100

int zahlen[SIZE]= {};
REAL werte[SIZE]= {};
```

- Bezeichner meist in GROSSBUCHSTABEN
- Makrodefinition endet bei Zeilenumbruch, mehr Zeilen mit `\`
- reine Textersetzung durch Präprozessor,
  daher keine Typsicherheit wie bei `typedef`
- Makros können auch Parameter haben:

```
#define DEBUG( cond, msg) if (cond) \
            cerr << "DBG: " << msg << endl;
...
DEBUG( nenner==0, "Nenner ist Null!");
```

- Aufhebung des Makros (undefine): `#undef SIZE`