C++ Teil 8

Sven Groß

7. Dez 2015

Themen der letzten Vorlesung

- Zeiger, Felder (Wdh.)
- dynamische Speicherverwaltung
- Cast bei Zeigern
- STL-Vektoren

Heutige Themen

1. Wdh.: Vektoren
2. Mehr zur STL
   - Sequentielle Container
   - Iteratoren
3. Ein-/Ausgabe über Dateien
4. Zusammengesetzte Datentypen
   - Strukturen (Structs)
   - Überladen von Operatoren

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 oder `at`
  Nummerierung 0, . . . , n-1, für `[ ]` 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!!
```

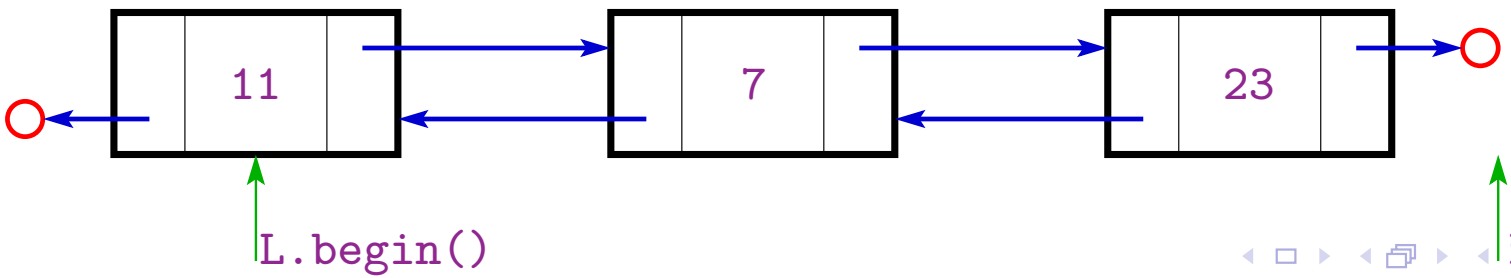
- Zuweisung `=` funktioniert wie erwünscht (im Gegensatz zu Feldern)
- kein Vektor im mathematischen Sinne:
  arithmetische Operatoren (`+, -, *`) nicht implementiert
- Ändern der Länge führt evtl. zu Reallozierung an anderer Speicheradresse
  $\rightsquigarrow$ Vorsicht mit Zeigern!
- STL definiert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

Sequentielle Container: Liste

- **Seq. Container** für linear angeordnete Daten (1 Vorgänger/Nachfolger)
- STL bietet z.B. `vector` (kennen wir schon), `list` (heute), `stack`, ...
- `list`: doppelt verkettete Liste
  - `#include<list>` einbinden
  - jedes Element enthält Zeiger auf Vorgänger/Nachfolger
  - Zugriff auf Elemente mittels **Iterator**

```
list<int> L;
L.push_back( 7);     L.push_back( 23);
L.push_front( 11);

for (list<int>::iterator it= L.begin(); it != L.end(); ++it)
    cout << *it << ",";
```



Liste (2)

Vergleich zu Vektor:

- — kein direkter Zugriff, z.B. auf 3. Element, möglich (kein *random access*)
- — mehr Speicherbedarf als Vektor
- \+ Einfügen/Löschen von Elementen aus der Mitte
- \+ Zusätzliche Funktionalität, z.B. `sort`, `merge`, `reverse`

```
    // Eintraege von L:  11 7 23
list<int>::iterator pos= L.begin();
++pos;                   // pos zeigt nun auf 7
L.insert( pos, 33);      // 11 33 7 23
                         // pos zeigt weiter auf 7
L.erase( L.begin() ); //     33 7 23

L.sort();                // 7 23 33
```

Iteratoren

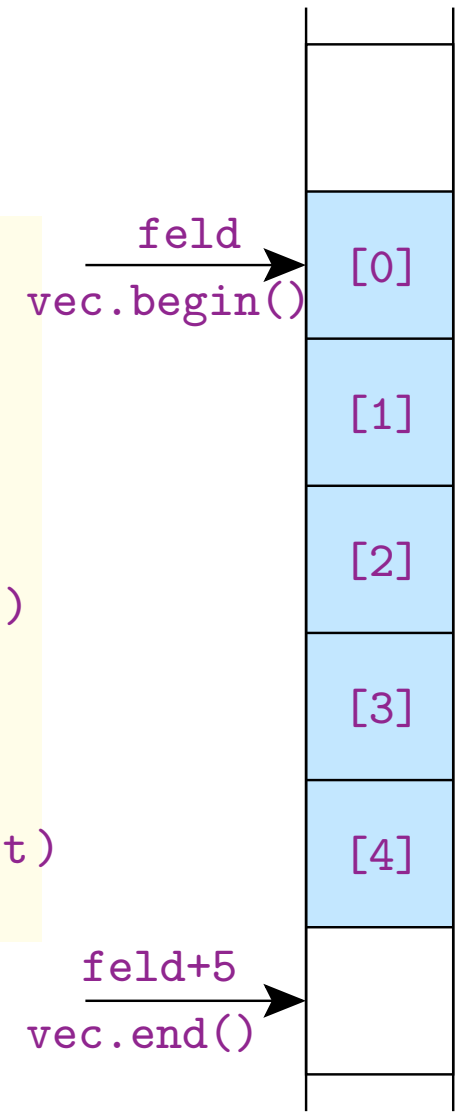
- werden in der STL benutzt, um Container zu durchlaufen
- verhalten sich wie Zeiger

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

vector<double> vec(5);
for (int i=0; i<5; ++i)
    vec[i]= feld[i];

for (double* zeig= feld; zeig != feld+5; ++zeig)
    cout << (*zeig) << endl;

for (vector<double>::iterator it= vec.begin();
                              it != vec.end(); ++it)
    cout << (*it) << endl;
```



- Datentyp: `ContainerT::iterator`
- Operatoren: `++`, `--`, `==`, `!=`, `*` (Dereferenzierung)

File Streams: Ein-/ Ausgabe über Dateien

- zuerst header-Datei für *file streams* einbinden:
  `#include <fstream>`
- Öffnen des *input* bzw. *output file streams* zur Ein- bzw. Ausgabe:

```
std::ifstream ifs;
ifs.open("mydata.txt");

std::ofstream ofs("results.dat");
```

- gewohnte Verwendung von Ein- und Ausgabeoperator
  (wie mit `cin` und `cout`)

```
ifs >> a;
ofs << "Das␣Ergebnis␣lautet␣" << f(x) << ".\n";
```

- bei Dateiausgabe: `endl` vermeiden, durch `\n` ersetzen (Pufferung)
- Schließen des *file streams* nach Verwendung:
  `ifs.close(); ofs.close()`

Basisklassen `istream`, `ostream`

- `cout` und `ofs` haben unterschiedliche Datentypen,
  aber gemeinsamen Basis-Datentyp: `std::ostream`
- `cin` und `ifs` haben unterschiedliche Datentypen,
  aber gemeinsamen Basis-Datentyp: `std::istream`
- Basis-Datentypen werden auch als **Basisklassen** bezeichnet,
  von denen speziellere Datentypen **abgeleitet** werden,
  z.B. `std::ofstream` ist abgeleitet von `std::ostream`

Beispiel:

```
void Ausgabe( double x, std::ostream& os)
{
    os << x << '\n';
}
```

kann benutzt werden als

- `Ausgabe( 3.14, cout);` → Bildschirmausgabe
- `Ausgabe( 3.14, ofs);` → Dateiausgabe

Einlesen per Eingabeoperator >>

- Eingabeoperator `>>` erwartet, dass Daten durch Whitespace getrennt sind (Leerzeichen, Tabulator, Zeilenumbruch)
- Abfrage des Stream-Status:
  - `ifs.good()` : Stream bereit zum Lesen
  - `ifs.eof( )` : Ende der Datei (*end of file*) erreicht
  - `ifs.fail()` : letzte Leseoperation nicht erfolgreich
  - `ifs.clear()`: setzt Status zurück, ermöglicht weiteres Lesen

⚠ Leseoperation `>>` wird nur bei `good`-Status durchgeführt

Beispiel:

```
ifstream ifs( "input.txt");
double d;  int i;  string s;

ifs >> d >> i;    // good
ifs >> i;         // fail
ifs.clear();      // good
ifs >> s;         // good
ifs >> s;         // fail eof
```

Datei `input.txt`:

```
17.5
4711 Hallo
```

Zeilenweises Lesen

- Whitespace-Trennung wie hier manchmal unerwünscht:

```
string vorname, nachname;
cout << "Bitte␣Vor-␣und␣Nachname␣eingeben:␣";
cin >> vorname >> nachname;
```

- Um eine ganze Zeile bis zum nächsten *Enter* (’\n’) zu lesen:

```
string name;
cout << "Bitte␣Vor-␣und␣Nachname␣eingeben:␣";
getline( cin, name);
```

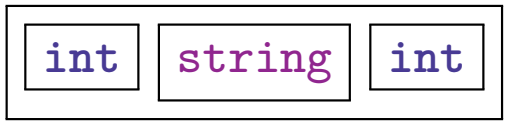
- funktioniert genauso für *input file streams*:

```
ifstream ifs( "fragen.txt"); // eine Zeile pro Frage
vector<string> question;

while ( !ifs.eof() ) {
    string zeile;
    getline( ifs, zeile);
    question.push_back( zeile); // hinten anhaengen
}
```

Strukturen (*Structs*)

- **Strukturen**: Zusammenfassung unterschiedlicher Datentypen zu einem *neuen benutzerdefinierten Datentyp*
- z.B. neuer Datentyp `Datum`:




```
struct Datum
{
    int    Tag;
    string Monat;
    int    Jahr;
}; // Semikolon nicht vergessen!
```

```
Datum MamasGeb, MeinGeb;

MamasGeb.Tag=   12;
MamasGeb.Monat= "Februar";
MamasGeb.Jahr=  1954;

MeinGeb= MamasGeb;  // Zuweisung
MeinGeb.Jahr+= 30;  // ich bin genau 30 Jahre juenger
```

Objekte

- Struktur-Variablen nennt man auch **Objekte**:
  `MeinGeb` ist ein Objekt vom Datentyp `Datum`.
- Zugriff auf **Datenelemente** eines Objektes mit '`.`' :
  `MeinGeb.Monat` ist ein Datenelement des Objektes `MeinGeb`.

```
cout << "Mein Geburtstag ist im " << MeinGeb.Monat;
```

Datenelemente nennt man synonym auch **Attribute**.

- Zugriff auf Datenelemente über Objektzeiger:

```
Datum *datzeig= &MeinGeb;

(*datzeig).Tag= 7;
cout << "Mein Geburtstag ist im " << (*datzeig).Monat;
```

einfachere Schreibweise mit `->` ist besser lesbar:

```
datzeig->Tag= 7;
cout << "Mein Geburtstag ist im " << datzeig->Monat;
```

Überladen von Operatoren (*operator overloading*)

- praktisch wäre, folgendes schreiben zu können:

```
if (MamasGeb < MeinGeb) // Datumsvergleich
    cout << "Mama ist aelter als ich.";
```

- möglich, wenn man folgende Funktion `operator<` definiert:

```
bool operator< (const Datum& a, const Datum& b)
{
    ...
}
```

Das nennt man **Operatorüberladung**.

- `MamasGeb < MeinGeb` ist dann äquivalent zum Funktionsaufruf `operator< ( MamasGeb, MeinGeb )`
- für viele Operatoren möglich, z.B. `+ - * / == >> <<` usw., je nach konkretem Datentyp sinnvoll

Überladen von Operatoren – Beispiel

Beispiel: Ein-/Ausgabeoperator für `Datum`:

- Wir wollen Code schreiben wie

```
ifstream ifs( "Datum.txt" );    // 3. Dezember 2014
Datum heute;
ifs >> heute;
cout << "Heute␣ist␣der␣" << heute << endl;
```

- Definition des **Ausgabeoperators** `<<` :

```
ostream& operator<< ( ostream& out, const Datum& d)
{
    out << d.Tag << ".␣" << d.Monat << "␣" << d.Jahr;
    return out;
}
```

- **Quiz:** Wie könnte der **Eingabeoperator** `>>` definiert werden?

```
istream& operator>> ( istream& in, Datum& d)
{
    // ???
}
```