technische universität dortmund

Einführung in die Programmierung

Wintersemester 2012/13

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

Inhalt

- Überblick über die **S**tandard **T**emplate **L**ibrary
- Datenstrukturen
- Exkurs: Iteratoren
- Exkurs: Konstante Objekte
- Praxis:
  - Function Objects
  - IO Manipulators
  - stringstreams
  - Operator << für eigene Klassen

Standard Template Library

- **Standard**: Verbindlich für alle Compiler
- **Template**: Große Teile sind als Templates implementiert

Besteht aus drei großen Teilen:
- Container / Datenstrukturen
- Input / Output
- Anderes: Algorithmen, Zufallszahlengenerator, etc

Rückblick

Wir haben bereits Teile der STL kennengelernt:

→ Kapitel 2: Namensraum `std` & `std::cout`

→ Kapitel 5: Funktionen der C Bibliothek

→ Kapitel 9: Die Klassen `std::string` & `std::fstream`

std::vector

`#include <vector>`

- Einfacher Container
- Wahlfreier Zugriff in konstanter Zeit (wie Array)
- Wächst dynamisch
- Speichert Kopien der Daten

```
using namespace std;
…
vector<int> zahlen;                                  // Leerer vector für int Variablen

// Erzeugt einen vector der bereits 30 mal den string „Leeres Wort“ enthält
vector<string> emptyWords(30, „Leeres Wort“);

for(int i=0; i < 49; ++i){
  zahlen.push_back(i*i);                             // Daten hinten anfügen
}

cout << zahlen[12] << endl;                          // Zugriff mit operator []
cout << zahlen.at(2) << endl;                        // Zugriff mit Methode at()
```

std::vector – Zugriff auf Daten

```
#include <vector>
```

- Wie bei Arrays über Indizes 0 … n-1
- Dank `operator[]` auch mit der gleichen Syntax
- Was ist der Unterschied zur Methode `at()`?

```
// Erzeugt einen vector der 20 mal die Zahl 42 enthält
vector<int> zahlen(20, 42);
cout << zahlen[10000] << endl;
```

Laufzeitfehler!
10000 ist kein gültiger Index. Programm stürzt ab.

- `operator[]` führt keine Bereichsüberprüfung durch (Effizienz!).
- Die Methode `at()` dagegen schon:

```
// Erzeugt einen vector der 20 mal die Zahl 42 enthält
vector<int> zahlen(20, 42);
try{
  cout << zahlen.at(10000) << endl;
} catch(out_of_range& ex){
  cout << "Exception: " << ex.what() << endl;
}
```

Funktioniert:
`at()` wirft eine Ausnahme, die wir dann fangen.

```
#include <stdexcept>
```


std::vector – Zugriff auf Daten

```
#include <vector>
```

- Beide Varianten geben Referenzen zurück
- Dadurch sind auch Zuweisungen möglich:

```
vector<int> zahlen;

zahlen.push_back(1000);
zahlen.push_back(2000);

zahlen[0] = 42;        // Überschreibt die 1000 mit 42
zahlen.at(1) = 17;     // Überschreibt die 2000 mit 17
```

Vorsicht!

- Zuweisungen nur an Indizes möglich, an denen schon Daten gespeichert waren!
- Neue Daten mit `push_back()` oder `insert()` einfügen
- `insert()` speichert ein Datum an einem vorgegebenen Index

std::vector – Zugriff auf Daten mit Iteratoren

```
#include <vector>
```

- Weitere Alternative für Datenzugriff
- Ein Iterator ist ein Objekt, das sich wie ein Pointer verhält
- Woher bekommt man Iteratoren? Z.B. von der Methode `begin()`:

```
vector<int>::iterator it = zahlen.begin();

while(it != zahlen.end()){   // Ende erreicht?
  cout << *it << endl;       // Dereferenzieren für Datenzugriff
  ++it;                      // Zum nächsten Element gehen
}
```

- Iteratoren können wie Pointer dereferenziert werden ⇒ so kommt man an die Daten
- Durch De-/Inkrement kommt man zu vorhergehenden oder nachfolgenden Daten

std::vector – Zugriff auf Daten mit Iteratoren

```
#include <vector>
```

Noch ein Beispiel: Wir wollen hochdimensionale reelle Daten speichern.
⇒ Ein Datum ist ein `std::vector<double>`
⇒ Die speichern wir in einem `std::vector`:

```
vector<vector<double> > data(100, vector<double>(30, 0.0));
```

Bei alten Compilern Leerzeichen nötig!

```
vector<vector<double> >::iterator it = data.begin();
```

Typ wird immer komplizierter – geht das nicht schöner?

Mit C++11: **JA mit Schlüsselwort `auto`**

```
auto it = data.begin();
```

- Platzhalter für Datentyp
- Überall dort erlaubt, wo der Compiler den benötigten Typ bestimmen kann

std::vector – Größe & Kapazität

`#include <vector>`

- `size()` liefert die Anzahl der gespeicherten Elemente:

```
for(int i=0; i < zahlen.size(); ++i){
  cout << zahlen[i] << „, “;          // Über alle Element iterieren
}
cout << endl;
```

- `capacity()` liefert den aktuell verfügbaren Speicherplatz:

```
cout << „Vector hat Platz für “ << zahlen.capacity() <<
  „ Elemente“ << endl;
```

- Reicht der Speicherplatz nicht mehr, wird mehr Platz bereitgestellt und vorhandene Daten werden umkopiert (Teuer!)
- Wenn vorher bekannt ist, wie viel Speicherplatz gebraucht wird, kann man diesen direkt reservieren:

```
vector <int> zahlen(1024);      // Platz für 1024 Elemente
```


std::vector – Praxis

`#include <vector>`

Überlegungen vorab:
- Wir wollen Daten aus der Astronomie bearbeiten
- Ein Datum besteht aus einem Index und 17 reellen Zahlen

```
// Minimal Variante, in Wirklichkeit “richtige” Klasse mit Kapselung und viel mehr Attributen
class StarData{
public:
  unsigned long long int index;
  double data[17];
};
```

- Beim Speichern in einem `vector` wird Kopie erzeugt (u.U. teuer!)

⇒ `vector` speichert Pointer auf dynamisch allokierte `StarData` Objekte

⇒ `vector` wird in eigener Klasse weggekapselt

std::vector – Praxis

`#include <vector>`

```
class GalaxyData{
public:
  GalaxyData(const string& filename);
  ~GalaxyData(){clear();}

  StarData* at(unsigned int i){return data.at(i);}

  void clear(){
    for(unsigned int i=0; i < data.size(); ++i){
      delete data[i];
    }
    data.clear();
  }

  unsigned int size(){return data.size();}

private:
  vector<StarData*> data;
};
```


std::vector – Praxis

`#include <vector>`

```
// Implementierung nicht korrekt – nur Idee!
GalaxyData::GalaxyData(const string& filename){
  ifstream file(filename);

  while(file.good() && !file.eof()){
    StarData* s = new StarData();

    string line;
    std::getline(file, line);

    s->index = atol(line.substr(0,
      line.find_first_of(",")).c_str());

    data.push_back(s);
  }

  file.close();
}
```

std::vector – Praxis

```
#include <vector>
```

```
int main(){
  GalaxyData dataSet("palomar5Selected.txt");

  cout << "Identifier des letzten Sterns: " <<
    dataSet.at(dataSet.size()-1)->index << endl;

  // Viel rechnen und analysieren

  dataSet.clear();
}
```


Wir kennen aus Kapitel 4 bereits konstante Variablen:

```
const double PI = 3.141;
char* const s3 = "Konstanter Zeiger auf char";
```

- Konstante Variablen dürfen nur initalisiert werden
- Jede weitere Zuweisung führt zu einem Compiler Fehler:

```
PI = 42.0;
```

Compilerfehler: error C3892: "PI": Einer Variablen, die konstant ist, kann nichts zugeordnet werden.

Was passiert bei Objekten, die als konstant deklariert wurden?

Beispiel: Minimalistische Klasse für zweidimensionale Punkte

```
class Point2D{
public:
  Point2D():_x(0),_y(0){}
  Point2D(double x, double y):_x(x), _y(y){}

  double getX(){return _x;}
  double getY(){return _y;}
  void setX(double x){_x = x;}
  void setY(double y){_y = y;}

private:
  double _x, _y;
};
```



```
int main(){
  Point2D p1(23, 89);
  const Point2D p2(-2, 3);

  p2 = p1;          // 1. Fehler: Zuweisung an konstantes Objekt ☑

  p2.setX(-1);      // 2. Fehler: Methodenaufruf mit konstantem Objekt ☑

  cout << "X Wert von p2: " << p2.getX()<< endl; // 3. Fehler: dito ☹

  return 0;
}
```

- Offenbar kann man für konstante Objekte keine Methoden aufrufen.
- Fehler 1 & 2 sind gewollt: Objekt `p2` ist als konstant deklariert ⇒ soll nicht verändert werden können!
- Fehler 3 ist frustrierend: `getX()` verändert das Objekt nicht, Aufruf sollte erlaubt sein!

⇒ Man muss dem Compiler mitteilen, welche Methoden für konstante Objekte aufgerufen werden dürfen!

```
class Point2D{
public:
  Point2D():_x(0),_y(0){}
  Point2D(double x, double y):_x(x), _y(y){}

  double getX() const {return _x;}
  double getY() const {return _y;}
  void setX(double x){_x = x;}
  void setY(double y){_y = y;}

private:
  double _x, _y;
};
```

Schlüsselwort `const` am Ende der Methodensignatur kennzeichnet Methoden, die für konstante Objekte aufgerufen werden dürfen.

```
int main(){
  const Point2D p2(-2, 3);

  cout << "X Wert von p2: " << p2.getX()<< endl;

  return 0;
}
```


Hinweise

- Nur solche Methoden mit `const` kennzeichnen, die das Objekt nicht verändern
- Man kann Methoden bezüglich `const` auch überladen, siehe z.B. `std::vector`:

std::vector::operator[]

```
      reference operator[] (size_type n);
const_reference operator[] (size_type n) const;
```

Warum konstante Objekte?

- Zusicherung, bei deren Überprüfung der Compiler hilft ⇒ nützlich
- Objekte bei Funktionsaufrufen zu kopieren ist teuer aber bei Übergabe per Referenz wären Änderungen außerhalb der Funktion sichtbar ⇒ mit konstanten Referenzen auf Objekte kann das nicht passieren

```
template<typename T>
void print(const vector<T>& v){
  // …
}
```


Viele weitere Datenstrukturen…

- `std::list` – entspricht unserem ADT Liste
- `std::queue` – entspricht unserem ADT Schlange (LIFO)
- `std::stack` – entspricht unserem ADT Stack (FIFO)

Praxis: Sortieren

```
#include <algorithm>
```

- std::sort erwartet optional eine Sortierfunktion oder ein Function Objekt
- Unterschied: Function Object erlaubt Parameter
- Was ist eigentlich ein Function Object? ⇒ Klasse, die operator() hat!

```
class VectorSorter{
  unsigned int _index;
public:
  VectorSorter(unsigned int index):_index(index){}

  bool operator()(const vector<int>& v1, const vector<int>& v2) const{
    return v1[_index] < v2[_index];
  }
};
```

```
int main(){
  vector<vector<int>> data;

  for(int i=0; i < 10; ++i){
    vector<int> v;
    for(int j=1; j <= 4; ++j){
      int zahl = std::rand();
      v.push_back(zahl);
    }
    data.push_back(v);
  }

  cout << "Unsortiert: " << endl;
  for(int i=0; i < 10; ++i){
    cout << "v" << i << ": " << data[i][0] << ", " << data[i][1] <<
    ", " << data[i][2] << ", " << data[i][3] << endl;
  }
  cout << endl;

  // Fortsetzung folgt…
```



```
  cout << "Nach erster Spalte sortiert: " << endl;
  std::sort(data.begin(), data.end(), VectorSorter(0));
  for(int i=0; i < 10; ++i){
    cout << "v" << i << ": " << data[i][0] << ", " << data[i][1] <<
    ", " << data[i][2] << ", " << data[i][3] << endl;
  }
  cout << endl;

  cout << "Nach letzter Spalte sortiert: " << endl;
  std::sort(data.begin(), data.end(), VectorSorter(3));
  for(int i=0; i < 10; ++i){
    cout << "v" << i << ": " << data[i][0] << ", " << data[i][1] <<
    ", " << data[i][2] << ", " << data[i][3] << endl;
  }

  return 0;
}
```


Einstellen von Parametern für Ausgabeströme

`#include <iomanip>`

⇒ Präzision bei **`float`**/**`double`**, Vorzeichen ja/nein, u.v.m.

```
int main(){
  cout << 5.123456789 << endl;
  cout << setprecision(2) << 5.123456789 << endl; // nur zwei stellen
  // vorzeichen bei positiven zahlen
  cout << setiosflags(ios_base::showpos) << 5.123456789 <<
    resetiosflags(ios_base::showpos) << " " << 5.123456789 << endl;
  // Ausgabe in 10 Zeichen breiten Spalten
  for(int i=0; i < 100); ++i){
    cout << std::setw(10) << std::rand();
    if((i+1) % 5 == 0){
      cout << endl;
    } else {
      cout << " ";
    }
  }
  return 0;
}
```


std::ostringstream

`#include <sstream>`

- Verhält sich wie Ausgabestrom **`cout`**
- Speichert die erzeugte Zeichenkette intern
- Besonders nützlich für GUI Programmierung (kommt demnächst)

```
class Point2D{
// Rest der Klasse wie vorhin
public:
  std::string toString() const;
};
```

```
std::string Point2D::toString() const{
  std::ostringstream result;
  result << "Point2D[" << _x << "," << _y << "]";
  return result.str();
}
```

```
Point2D p(-2.0, 3.9);
cout << p.toString() << endl;
guiWindow.setStatusbarText(p.toString()); // guiWindow = fiktives GUI
```

Implementierung von operator<<

Überlegungen vorab:

- Für primitive Datentypen sind die Operatoren Teil der Stream Klassen:

```
cout << 20 << „/“ << 5 << „=„ << (20/5);
// entspricht:
cout.operator<<(20).operator<<(„/“).operator<<(5).operator<<(„=„).
  operator<<((20/5));
```

- Operatoren für eigene Datentypen (Klassen)? ⇒ IDEE: Neue Stream Klasse von ostream ableiten, Operatoren hinzufügen
- ABER: Laufzeitbibliothek instanziert für cout nicht unsere abgeleitete Klasse ☹
- Nächste Idee: Methode in der Klasse, die wir ausgeben wollen? ⇒ Geht auch nicht (siehe oben)
- Ausweg: Globale Funktion!

Implementierung von operator<< - Point2D Klasse

```
class Point2D{
public:
  Point2D():_x(0),_y(0){}
  Point2D(double x, double y):_x(x), _y(y){}

  double getX() const {return _x;}
  double getY() const {return _y;}
  void setX(double x){_x = x;}
  void setY(double y){_y = y;}

private:
  double _x, _y;
};

std::ostream& operator<<(ostream& s, const Point2D& p){
  s << "Point2D[" << p.getX() << "," << p.getY() << "]";
  return s;
}
```


Implementierung von operator<< - Point2D Klasse

```
int main(){
  Point2D p1(23, 89);

  cout << p1 << endl;

  return 0;
}
```

Resultat: