technische universität
dortmund

Einführung in die Programmierung

Wintersemester 2020/21

Kapitel 14: Standard Template Library

M.Sc. Roman Kalkreuth
Lehrstuhl für Algorithm Engineering (LS11)
Fakultät für Informatik

Inhalt

- Überblick über die **S**tandard **T**emplate **L**ibrary
- Datenstrukturen
- Exkurs: Iteratoren
- Exkurs: Konstante Objekte
- Praxis:
  - Function Objects

Standard Template Library (STL)

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

Besteht aus drei großen Teilen:
- Container / Datenstrukturen
- Input / Output
- Sonstiges: 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
                              // Speicherallokation automatisch
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 (oder Schlimmeres – Verhalten undefiniert!).

- **`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? Zum Beispiel über die 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>`**
⇒ Mehrere von diesen double-Vektoren speichern wir in einem **`std::vector`**:

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

vor C++11 Leerzeichen notwendig

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

Typ wird immer komplizierter – geht das nicht schöner?

Seit 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 Elemente 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
```

Exkurs: Konstante Objekte

Wir kennen aus Kapitel 4 bereits konstante Variablen:

```
double const PI = 3.1415;
char const *const s3 = "Konstanter Zeiger auf konstantes char";
```

- Konstante Variablen dürfen nur initialisiert werden
- Jede weitere Zuweisung führt zu einem Compilerfehler:

```
PI = 42.0;
```

Compilerfehler:
error: assignment of read-only variable ‘PI’

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;
};
```

Exkurs: Konstante Objekte

```
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.

Exkurs: Konstante Objekte

```
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;
};
```

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

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

  return 0;
}
```

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

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 **Referenzen auf konstante Objekte** kann das nicht passieren!

```
template<typename T> void print(vector<T> const& v){
  // Diese Funktion "verspricht" schon in der Schnittstelle, dass
  // sie das (per Referenz) übergebene Objekt nicht verändert.
  // Der Compiler überprüft dieses Versprechen.

  // ...
}
```

Viele weitere Datenstrukturen …

- **`std::list`** – entspricht unserem ADT Liste
- **`std::queue`** – entspricht unserem ADT Schlange (LIFO)
- **`std::stack`** – entspricht unserem ADT Stack (FIFO)
- **`std::map`** – Abbildung **`key`** → **`value`**, wobei **`key`** ∈ beliebiger sortierbarer Index

realisiert ADT binären Suchbaum

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 {
private:
  unsigned int _index;

public:
  VectorSorter(unsigned int index) : _index(index) {}

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

⇒ zum Sortieren von Vektoren gemäß Vektorkomponente **`index`**

Function Objects - Sortieren

```
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 …
```

einfache(re), aber unflexiblere Zufallszahlenquelle

```
 // Fortsetzung:

  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;
  }
}
```