Programmieren in C++

SS 2018

Vorlesung 9, Dienstag 26. Juni 2018

(STL, Sortieren)

Axel Lehmann
Lehrstuhl für Algorithmen und Datenstrukturen
Institut für Informatik
Universität Freiburg

UNI FREIBURG

Blick über die Vorlesung heute

UNI FREIBURG

- Organisatorisches
  - Erfahrungen mit dem Ü8 — Templates
  - Keine Studium Generale — Dafür zwei kleine Exkurse
  - Erste Infos zum Projekt — FAQ
- Inhalt
  - Standard Template Library — std::string, std::vector, …
  - Sortieren mit der STL — std::sort
  - **Übungsblatt 9:**
    Die häufigsten Zeichenketten in einem gegebenen Text berechnen … mit Hilfe vieler Funktionen aus der STL

Erfahrungen mit dem Ü8

UNI FREIBURG

- Zusammenfassung / Auszüge
  - Für die meisten sehr gut und zügig machbar
  - A1 Probleme mit Pointern und Speicherverwaltung speziell bei Linked-List
    Dazu gleich mehr
  - „leider lassen sich insert und erase nicht testen, bevor lookup bzw. lookup und insert funktioniert."
    Es dürfen immer zusätzliche Tests geschrieben werden
  - „Das mit den Bitmasken fand ich interessant, hat Spaß gemacht"

Exkurse 1/2

UNI FREIBURG

- Linked-List
  - lookup O(n)
  - insert O(1)
  - erase O(1)
- Dynamic Array
  - lookup O(n)
  - insert O(n)
    amortisiert O(1)
  - erase O(n)
    amortisiert O(1)

Exkurse 2/2

UNI FREIBURG

- Plagiate
  - Aus gegebenem Anlasse:

    "Offensichtliches Plagiat": Bethesda verklagt Warner wegen Westworld-Mobilspiel (heise.de)

Projekt

UNI FREIBURG

- Frequently Asked Questions
  - Wann ist die Abgabefrist?

    Dienstag, 18. September um 12:00 Uhr
  - Muss man das Projekt machen?

    Ja
  - Auch wenn man sonst schon genug Punkte hat?

    JA
  - Ist für das Projekt Gruppenarbeit erlaubt?

    Nein

    Vorlesung 11 wird fast nur über das Projekt gehen, alle weiteren Infos dann dort

STL 1/13

UNI FREIBURG

- STL = Standard Template Library
  - Die C++ Standardbibliothek
  - Nützliche Klassen und Methoden, die man immer wieder in den verschiedensten Anwendungen braucht, z.B.

std::string	(Zeichenketten)
std::vector	(dynamische Felder)
std::map	(assoziative Felder, sortiert)
std::unordered_map	(assoziative Felder, unsortiert)

  - Dank templates können viele Klassen für alle möglichen Arten von Objekten benutzt werden können, z.B.

    std::vector<Trader>, std::map<std::string, size_t>, …

STL 2/13

UNI FREIBURG

- Namespaces, Motivation
  - Ein **namespace** ist einfach ein zusätzlicher Namenspräfix
  - Um den Quelltext der STL steht dazu

    namespace std { … }
  - Um eine Klasse daraus zu benutzen, muss man dann std:: davor schreiben, z.B. std::vector<int>

    Dann gibt es keine Verwechslungsgefahr, wenn man selber z.B. auch eine Klasse vector schreibt

STL 3/13

UNI FREIBURG

- Namespaces, Benutzung
  - Wenn man eine Klasse sehr oft benutzt und nicht jedes Mal std:: davor schreiben will, kann man auch schreiben

    ```
    using std::vector;  // "Declare" usage of std::vector
    …
    vector<int> v;      // Now I can use it without std::
    ```

  - Theoretisch könnte man auch schreiben

    ```
    using namespace std; // Use complete STL without std::
    ```

    Aber das erlaubt checkstyle nicht, weil man damit das ganze namespace Konzept wieder "aushebelt"

UNI FREIBURG

- Input / Output (Beispiele)
  - C Style (stdio):

```
#include <stdio.h>

printf("Doof\n");
fprintf(stderr, "Bloed\n");
FILE* file = fopen("bloed.txt", "r");
```

  - C++ Style (STL):

```
#include <iostream>
#include <fstream>

std::cout << "Doof" << std::endl;
std::cerr << "Bloed" << std::endl;
std::ifstream file("bloed.txt");
```

UNI FREIBURG

- Die Klasse **std::string**
  - Eine komfortable Klasse für Zeichenketten, z.B.

```
#include <string>

std::string s = "doov";
size_t pos = s.find("v");
if (pos != std::string::npos) { s[pos] = 'f'; }
printf("%s\n", s.c_str());  // Prints "doof".
```

  - Intern speichert die Klasse einen null-terminierten C-String, und den bekommt man mit der c_str() Methode

    Nützlich, um C-Methoden auf einem std::string zu nutzen

  - Häufig benutzte Methoden (Details siehe Links am Ende):

    size, append, erase, substr, find, find_first_of, ...

UNI FREIBURG

- Die Klasse **std::ostringstream**
  - Kann genauso benutzt werden wie std::cout, mit <<

```
#include <sstream>
std::ostringstream oss;
oss << 5 << " x doof" << std::endl;
printf("%s\n", oss.str().c_str());  // Prints "5 x doof".
```

Das Äquivalent in Java ist der StringBuilder

  - Komfortables testen eigener Klassen mit ASSERT_STREQ

```
ASSERT_STREQ("[1, 2, 3, 4]", array.toString().c_str());
```

STL 6/14

UNI FREIBURG

- Beispielcode

```
#include <sstream>
...
std::string Array<T>::toString() {
  std::ostringstream oss;
  oss  <<  "[";
  for (size_t  i = 0; i < _size; i++) {
    oss  <<  (i > 0 ? ", " : "")  <<  _elements[i];
  }
  oss  <<  "]";
  return oss.str();  // The assembled string;
}
```

UNI FREIBURG

- Die Klasse **std::vector**
  - Für dynamische Felder (= können Ihre Größe beliebig ändern) von Objekten von einem beliebigen Typ

```
#include <vector>

std::vector<std::string> v;          // Empty array
v.push_back("doof");                 // Append one element.
v.push_back("bloed");                // Another one.
for (size_t  i = 0; i < v.size(); i++)
  { printf("%s\n", v[i].c_str()); }  // Print i-th element.
```

  - Häufig benutzte Methoden (Details siehe Links am Ende):

    size, push_back, pop_back, resize, begin, end, ...

- Die Klasse **std::vector**, Initialisierung
  - Seit **C++11** Standard kann man std::vector wie ein statisches C-Feld initialisieren, z.B.

```
std::vector<int> v1 { 5, 1, 4, 3, 2 };

std::vector<std::string> v2 { "doof", "bloed" };
```

- Iteration über die Elemente eines **std::vector**
  - Geht statt mit einer for Schleife auch so:

```
std::vector<int> v { 5, 1, 4, 3, 2 };
for (std::vector<int>::iterator it = v.begin();
    it != v.end(); it++) { printf("%d\n", *it); }
```

Das "it" ist dann wie ein Zeiger auf das jeweilige Element

  - Alternativ kann man seit C++11 auch schreiben

```
std::vector<int> v { 5, 1, 4, 3, 2 };
for (auto& x : v) { printf("%d\n", x); }
```

Das x ist dann analog zu dem *it oben, und durch das "auto" wird der passenden Typ vom Compiler gewählt

UNI FREIBURG

- Die Klasse **std::map**
  - Für assoziative Felder (siehe Informatik II), z.B.

```
#include <map>

std::map<std::string, int> died;
died["W. Shakespeare"] = 1616; // Assign like array.
died["H. P. Lovecraft"] = 1937;   // Dito.
int x = died["H. P. Lovecraft"];    // Access like an array.
```

  - Häufig benutzte Methoden (Details siehe Links am Ende):

```
died.size()                        // Num. stored entries.
died.count("D. Adams") > 0  // Does entry exist in the
                                   // map?
```

UNI FREIBURG

- Die Klasse **std::map,** Achtung
  - In C++ gibt es folgende Besonderheit bei der map
  - Wenn man mit […] auf einen Schlüssel zugreift, der noch gar nicht in der map ist, wird dieser automatisch angelegt, und zwar mit dem Defaultwert
  - Das ist oft nützlich

```
counter["doof"]++;  // If entry exists, increase by one;
                    // otherwise first create with value 0
```

  - Kann aber zu unerwarteten Effekten führen

```
if (died["J. W. von Goethe"] < 1616) { … }
```

Falls es gar keinen Eintrag für "J. W. von Goethe" gab, gibt es nach dieser Anweisung einen, mit Wert 0

UNI FREIBURG

- Iteration über die Elemente einer **std::map**
  - Geht seit C++11 ähnlich wie beim std::vector

```
std::map<std::string, int> died;
died["W. Shakespeare"] = 1616;
died["H. P. Lovecraft"] = 1937;
...
for (auto& pair : died) {
   printf("Name: %s\n", pair.first.c_str());
   printf("Year: %d\n", pair.second);
}
```

Wie vorher wird durch das "auto" der Typ von "pair" automatisch passend vom Compiler gewählt

UNI FREIBURG

- Die Klasse **std::unordered_map**
  - Sehr ähnlich zu std::map, aber hält die Elemente nicht zu jedem Zeitpunkt sortiert nach den Schlüsseln

    Das ist deutlich effizienter, wenn man das nicht braucht

    Es ist auch effizienter, wenn man die Elemente nur einmal am Ende sortiert haben will

    Dann besser std::unordered_map, am Ende dann std::sort o. Ä. anwenden

    So sollen Sie es auch für das Ü9 machen

Sortieren mit der STL 1/4

UNI FREIBURG

- Sortieren von Objekten von beliebigem Typ
  - Beispiel

```
#include <algorithm>
#include <vector>

std::vector<int> v = { 5, 1, 4, 3, 2 };
std::sort(v.begin(), v.end());
```

  - Ohne weiteres Argument wird einfach der Operator **<** auf dem Elementtyp benutzt, in dem Fall auf int
  - Die Elemente werden in v umgeordnet ("in place")

    Im Beispiel oben stehen in v also nachher die Elemente 1, 2, 3, 4, 5 in der Reihenfolge

Sortieren mit der STL 2/4

UNI FREIBURG

- Die C++11 Variante
  - C++11 erlaubt sogenannte **lambda-Ausdrücke**, das sind einfach anonyme temporäre Funktionen, z.B.

```
std::vector<int> v { 5, 1, 4, 3, 2 };
std::sort(v.begin(), v.end(),
   [](const int& x, const int& y) { return x > y; } );
```

    Die Vergleichsfunktion wird an Ort und Stelle definiert (und ist auch nur für die Dauer des Sortierens gültig)

    Ergebnis und Effizienz sind identisch wie mit einer extra Klasse mit Vergleichsoperator, nur in dem Fall viel einfacher

  - std::sort(v.begin(), v.end(),
    std::greater<int>()); // Found in <functional>

Sortieren mit der STL 3/4

UNI FREIBURG

- Sortieren mit eigener Vergleichsfunktion
  - Beispiel (Code oben in .h Datei, Code unten in .cpp Datei)

```
class MyComparison {
   // Return true iff x comes before y in desired order.
   public: bool operator()(const int& x, const int& y) {
      return x > y;          // Larger number wins now.
   }
};
...
std::vector<int> v { 5, 1, 4, 3, 2 };
MyComparison cmp;      // Object from the class above.
std::sort(v.begin(), v.end(), cmp);
```

Inhalt von v danach 5, 4, 3, 2, 1 in der Reihenfolge

Sortieren mit der STL 4/4

UNI FREIBURG

- Warum eine extra Klasse zum Vergleichen?
  - In der std::sort Methode wird immer wenn zwei Elemente x und y verglichen werden sollen cmp(x, y) aufgerufen

    Das ruft gerade die Methode MyComparison::operator() mit den Argumenten x und y auf
  - Auf der vorherigen Folie ist die Methode **inline** definiert, das heißt gleich bei der Deklaration in der .h Datei
  - Dann setzt der Compiler an die Stelle des Aufrufes cmp(x, y) gleich den Code aus der Funktion, in dem Fall x > y
  - Das spart im Maschinencode einen (teuren) Funktionsaufruf, das heißt: das Hin- und Zurückspringen im Code

Algorithmen in der STL 1/2

UNI FREIBURG

- in-place / copy
  - Die meisten Funktionen arbeiten in-place, d.h. Speicherplatz optimiert, z.B. sortieren

```
std::vector<int> v { 5, 1, 4, 3, 2 };
std::sort(v.begin(), v.end());
```

  - Manchmal möchte man Teile aber anderweitig verändern, daher gibt es die _copy Varianten

Algorithmen in der STL 2/2

UNI FREIBURG

- std::vector<int> v { 5, 1, 4, 3, 2 };
  std::vector<int> even; even.resize(v.size());
  std::vector<int> odd; odd.resize(v.size());

  auto limits = **std::partition_copy**(
  v.begin(), v.end(), even.begin(), odd.begin(),
  [](const int& a) { return a % 2 == 0; });

  for (size_t = 0;
  i < **std::distance**(odd.begin(), limits.second);
  ++i) { std::cout << odd[i] << std::endl; }

Literatur / Links

- STL und alles was dazu gehört
  - http://www.cplusplus.com/reference/stl
  - http://www.cplusplus.com/reference/string
  - http://www.cplusplus.com/reference/vector
  - http://www.cplusplus.com/reference/map
  - http://www.cplusplus.com/doc/tutorial/namespaces
  - http://www.cplusplus.com/reference/algorithm/sort
  - http://www.cplusplus.com/reference/iostream
  - http://www.cplusplus.com/reference/fstream
  - ...