Programmieren in C++

SS 2022

Vorlesung 9, Dienstag 28. Juni 2022

(Die STL = Standard Template Library)

Prof. Dr. Hannah Bast
Professur für Algorithmen und Datenstrukturen
Institut für Informatik, Universität Freiburg

UNI FREIBURG

Blick über die Vorlesung heute

UNI FREIBURG

- Organisatorisches
  - Erfahrungen mit dem Ü8 — Templates
  - Anmeldung Studienleistung — Deadline 30.07.2022
- Inhalt
  - SLT Datenstrukturen — vector, string, map, …
  - STL Ein- und Ausgabe — iostream, oftream, ifstream
  - STL Zeitmessung — chrono
  - STL Algorithmen — sort, lambdas

Ü9: Geodaten aus OSM als psychedelische HeatMap malen

Sie können dabei jetzt die STL benutzen … und unser TerminalManager kommt auch nochmal zur Geltung

Erfahrungen mit dem Ü8

UNI FREIBURG

- Rückmeldung zur Aufgabe
  - Templates waren sehr interessant und sehr gut erklärt

    Meinen ganz herzlichen Dank an dieser Stelle an Johannes für die beiden hervorragenden Vertretungen und die Mühe

    "Ich liebe es, mit Bit-Operatoren zu spielen!" [viele]

    "Es war nicht sonderlich schwer, aber ich habe viel gelernt"

    "Aufgaben waren gut machbar, freue mich trotzdem wenn wieder etwas kreativere Aufgaben lösen dürfen"

    "Sehr hilfreich, dass die Fehlermeldungen langsamer und ausführlicher erklärt wurden"

    "Wenn es nach mir ginge, könnten die Übungen tatsächlich doch wieder etwas umfangreicher ausfallen"

STL Einführung 1/4

UNI FREIBURG

- STL = Standard Template Library
  - Die STL ist die C++ Standardbibliothek
  - Enthält nützliche Klassen und Methoden, die man immer wieder in den verschiedensten Anwendungen braucht

    Rest der Vorlesung: ausgewählte Beispiele von Klassen und Methoden, die man in der Praxis sehr häufig braucht
  - Dank Templates können viele Klassen für alle Typen benutzt werden (sowohl Basistypen als auch Klassen)

    Ziel der STL ist dabei, möglichst genauso effizient zu sein wie der entsprechende C-Code

    Und meistens gelingt das auch, siehe dazu Folie 25

STL Einführung 2/4

UNI FREIBURG

- Warum die Standardbibliothek erst jetzt in V9
  - Um die STL auch in schwierigeren Anwendungsfällen effizient und korrekt benutzen zu können, sollte man verstehen, wie Sie im Prinzip programmiert ist

    Basiert stark auf Templates → V8

    Benutzt intensiv copy und move Semantik → V6 und V7

    Erfordert grundlegendes Verständnis von Klassen → V5

    Erfordert grundlegendes Verständnis von Zeigern → V4

    Es kommt bei der Benutzung der STL außerdem schnell zu komplexeren Fehlermeldungen, die man einfach nicht versteht, wenn man die zugrundeliegenden Konzepte nicht verstanden hat

STL Einführung 3/4

UNI FREIBURG

- Namespaces
  - Alle Klassen und Methoden der STL sind in einem sogenannten **namespace** definiert, mit Namen **std**

```
namespace std {
  // The fully qualified name of everything inside of this
  // namespace is std::…
}
```

  - Die Klasse **string** aus der STL heißt dann zum Beispiel mit vollem Namen **std::string**

    So gibt es keine Verwechslungsgefahr, falls eine andere Bibliothek oder man selber auch eine Klasse **string** definiert

  - Die STL benutzt außerdem (wie in C) Kleinschreibung und Unterstriche anstatt CamelCase, zum Beispiel std::string_view

STL Einführung 4/4

UNI FREIBURG

- Namespaces, Benutzung
  - Man kann entweder die voll qualifizierten Namen verwenden

```
#include <string>

std::string s = "doof";
```

  - Oder man "verspricht" einmal, dass keine Verwechslungs-gefahr für einen bestimmten Namen besteht und dann kann man ihn ohne namespace-Präfix benutzen

```
#include <string>

using std::string;

string s = "doof";  // No std:: needed thanks to using.
```

STL Zeitmessung

UNI FREIBURG

- Geht so

```
#include <chrono>

auto timeBeg = std::chrono::steady_clock::now();
...
auto timeEnd = std::chrono::steady_clock::now();
auto duration = timeEnd – timeBeg;
auto durationMsecs = std::chrono::duration_cast
                          <std::chrono::milliseconds>(duration);
int msecs = durationMsecs.count();
```

Man sieht hier schön, dass Abstraktion oft auch einen Preis hat in Form von Verbosität (das ist allerdings ein Extrembeispiel)

Normalerweise packt man diesen Code in eine Klasse und so machen wir es auch gleich beim Live-Coding

STL Ein- und Ausgabe 1/3

UNI FREIBURG

- Ausgabe auf standard input und standard error
  - C Style (stdio):

```
#include <stdio.h>

printf("doof\n");
fprintf(stderr, "bloed\n");
```

  - C++ Style (STL):

```
#include <iostream>

std::cout << "doof" << std::endl;
std::cerr << "bloed" << std::endl;
```

    std::endl ist das natürliche "newline" der Maschine

    Es sorgt auch dafür, dass **nicht gepuffert**, sondern tatsächlich herausgeschrieben wird → das geht auch explizit mit std::flush

STL Ein- und Ausgabe 2/3

UNI FREIBURG

- Überladen des << Operators
  - Bei eigenen Klassen möchte man oft definieren, wie ein Objekt aussehen soll, wenn man es in einen Ausgabe-Stream schreibt

```
std::ostream& operator<<(std::ostream& os, Timer& timer) {
  os << timer.msecsPassed() << " ms";
  return os;
};
```

Erklärung: Der Operator << wird als Infix-Operator benutzt, d.h. das erste Argument steht davor und das zweite dahinter

```
std::cout << timer;   // Calls operator<<(std::cout, timer)
```

Das erste Argument wird zurückgegeben, damit man bequem Folgen von << schreiben kann

```
std::cout << "Time spent: " << timer << std::endl;
```

STL Ein- und Ausgabe 3/3

UNI FREIBURG

- Lesen aus einer Datei
  - C Style (stdio):

```
#include <cstdio>

FILE* file = fopen("doof.txt", "r");  // Open for reading.
if (file == nullptr) { printf("Doof!\n"); exit(1); }
char line[1001];                      // Room for 1000 bytes + final \0
std::fgets(line, 1000, file);  // Read line, but at most 1000 bytes.
```

  - C++ Style (STL):

```
#include <ifstream>

std::ifstream file("doof.txt");
if (!file) { std::cout << "Doof!" << std::endl; exit(1); }
std::string line;
std::getline(file, line);   // Read line, resizes string as needed.
```

STL Datenstrukturen 1/8

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.
std::cout << v[1] << std::endl;      // Prints "bloed".
```

  - Häufig benutzte Methoden (Details siehe Referenzen):

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

STL Datenstrukturen 2/8

UNI FREIBURG

- Die Klasse **std::vector**, Initialisierung
  - Seit C++11 kann man einen std::vector wie ein statisches C-Feld initialisieren, zum Beispiel

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

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

  - Iterieren über einen std::vector geht bequem so:

```
std::vector<int> v = { 5, 1, 4, 3, 2 };
for (const auto& x : v) { std::cout << x << std::endl; }
```

    Erklärung: bei **auto** wählt der Compiler automatisch den passenden Typ, int in diesem Fall; man muss sich aber explizit entscheiden, ob als Referenz (auto&) oder als Kopie (auto), und ob const oder nicht

STL Datenstrukturen 3/8

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'; }
std::cout << s << std::endl;   // 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, ...

STL Datenstrukturen 4/8

UNI FREIBURG

- Die Klasse **std::string_view**
  - Oft möchte man auf Teile einer Zeichenkette als eigene Zeichenkette zugreifen, ohne dafür extra eine Kopie anlegen zu müssen, das geht seit C++17 mit std::string_view

```
#include <string>
#include <string_view>

std::string s = "Gar nicht so doof diese STL";
std::string_view sv = s;   // Does not copy the string to sv,
                           // but sv is just a "view" on s
sv.remove_prefix(13);      // sv is now just the part of s
                           // starting at the 14-the character

std::cout << sv.substr(0, 4) << std::endl;   // Prints "doof".
```

STL Datenstrukturen 5/8

UNI FREIBURG

- Die Klasse **std::map**
  - Für assoziative Felder (siehe Vorlesung "Algorithmen und Datenstrukturen"), zum Beispiel:

```
#include <map>

std::map<std::string, int> born;
born["Marie Curie"] = 1867;        // Assign like for an array.
born["Albert Einstein"] = 1879;
```

Seit C++17 kann man wie folgt über die Paare iterieren:

```
for (const auto& [name, year] : born) {
  std::cout << name << " was born in " << year << std::endl;
}
```

Man beachte wieder das **auto** in Kombination mit **const** und **&**, da wir die strings weder kopieren noch verändern möchten

STL Datenstrukturen 6/8

UNI FREIBURG

- Die Klasse **std::map,** Achtung
  - Wenn man in C++ mit […] auf einen Schlüssel zugreift, der noch gar nicht in der map ist, wird dieser automatisch angelegt, und zwar mit dem Defaultwert
  - Das hat folgenden nützlichen use case:

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

  - Es kann aber auch zu unerwarteten Effekten führen:

```
std::string name = getName();
if (born[name] < 1900) { std::cout << "Dead already!"; }
```

Falls es den Namen vorher nicht gab, gibt es ihn nach der zweiten Zeile, und zwar mit Wert 0; wenn man das öfter macht, hat man lauter unerwartete Einträge in seiner map

STL Datenstrukturen 7/8

UNI FREIBURG

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

    Implementierung: die std::map ist in der Regel als **Baum** implementieren, die std:unordered_map als **Hashtabelle**
  - Wenn man die Elemente nicht zu jedem Zeitpunkt sortiert braucht, ist die std::unordered_map viel effizienter

    Wenn man erst viele Element einfügen und dann einmal am Ende sortieren möchten, würde man auch eine std::unordered_map benutzen und dann std::sort am Ende

    Für das Ü9 ist eine std::unordered_map sinnvoll; wenn Sie eine Hashtabellenallergie haben, geht aber auch std::vector

STL Datenstrukturen 7/8

UNI FREIBURG

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

    Implementierung: die std::map ist in der Regel als **Baum** implementieren, die std:unordered_map als **Hashtabelle**
  - Wenn man die Elemente nicht zu jedem Zeitpunkt sortiert braucht, ist die std::unordered_map viel effizienter

    Wenn man erst viele Element einfügen und dann einmal am Ende sortieren möchten, würde man auch eine std::unordered_map benutzen und dann std::sort am Ende

    Für das Ü9 ist eine std::unordered_map sinnvoll; wenn Sie eine Hashtabellenallergie haben, geht aber auch std::vector

STL Datenstrukturen 8/8

UNI FREIBURG

- Eigene Hashfunktionen definieren
  - Die std::unordered_map ist mit zwei Typen templatisiert:

    K für den Schlüssel ("key") und V für den Wert ("value")
  - Um die key-value Paare in der Hashtabelle speichern zu können, braucht man eine Hashfunktion für den Typ K

    Siehe Vorlesung "Algorithmen und Datenstrukturen"
  - Für viele Basistypen (int, float, std::string, …) hat die STL Hashfunktionen definiert, für eigene Klassen muss man es selber machen, das geht zum Beispiel so

```
struct std::hash<std::pair<float, float>> {
  size_t operator()(const std::pair<float, float>& p) {
    return std::hash(p.first) + std::hash(p.second); }
};
```

STL Algorithmen 1/5

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 benutzt std::sort einfach den Operator **<** auf dem Elementtyp, in dem Fall auf int
  - Die Elemente werden in v umgeordnet ("in place")

    In dem Beispiel oben enthält also v nach dem Sortieren die folgenden Elemente: 1, 2, 3, 4, 5 (in der Reihenfolge)

STL Algorithmen 2/5

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 = { 1, 2, 3, 4, 5 };
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

STL Algorithmen 3/5

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

    Das messen wir gleich selber mal nach

STL Algorithmen 4/5

UNI FREIBURG

- Seit C++11 geht es auch einfacher
  - Seit C++11 gibt es sogenannte **lambda-Ausdrücke**, das sind (im Wesentlichen) anonyme temporäre Funktionen, z.B.

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

    Ein lambda entspricht genau einer Klasse mit operator() wie zwei Folien zuvor, es ist nur viel komfortabler zu definieren

  - Es gibt auch vordefinierte Funktionen (#include <functional>)

```
std::sort(v.begin(), v.end(), std::greater<int>());
```

STL Algorithmen 5/5

UNI FREIBURG

- Effizient von "inlining"
  - Bei einfachen Funktionen, die oft aufgerufen werden, macht es von der Effizienz her einen riesigen Unterschied, ob man es schafft, den Funktionsaufruf zu vermeiden
  - Das wollen wir uns jetzt mal an einem einfachen Codebeispiel in vier Varianten anschauen und nachmessen:

    V0: Kein Funktionsaufruf, Code steht direkt in der Funktion

    V1: Expliziter Funktionsaufruf

    V2: Klasse mit der Funktion als Template-Argument

    V3: Wie V2, aber mit lambda-Ausdruck statt Klasse

Hinweise zum Ü9

UNI FREIBURG

- TerminalManager
  - Beim Ü9 sollen Sie wieder etwas auf den Bildschirm malen
  - Sie können dazu eine leicht erweiterte Version der Klasse TerminalManager vom Ü5 verwenden, siehe Wiki und SVN

    Die Methoden **drawPixel** und **drawString** haben jetzt ein zusätzliches Argument "intensity" (Wert zwischen 0 und 1), mit der man in verschiedenen Schattierungen malen kann

    Das schauen wir uns jetzt mal noch zusammen anhand von einem einfachen Beispiel an

Referenzen

UNI FREIBURG

- Die 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
  - …