



Programmieren 3

C++

Vorlesung 10: Iteratoren und Streams

Prof. Dr. Dirk Kutscher
Dr. Olaf Bergmann

Terminplan

Abgabe Übung 3

30.11.2023: Vorlesung 10

07.12.2023: Vorlesung entfällt

14.12.2023: Das Semester im Schnelldurchlauf

19.12.2023: Abgabe Übung 4

21.12.2023: Probeklausur

04.01.2024: Fragen und Antworten

12.01.2024: Klausur (T-Foyer)

Lehrevaluation

Vorlesung



<https://evasys.hs-emden-leer.de/evasys/online.php?pswd=AXVYM>

Praktikum



<https://evasys.hs-emden-leer.de/evasys/online.php?pswd=PNHLF>

Wiederholung

Verbessertes sort

```
template<class Iterator, class Compare>
void mySort(Iterator first, Iterator last, Compare comp) {

    Iterator prev = first, next = first;

    // prev != last sicherstellen, bevor next inkrementiert wird
    if (prev == last || ++next == last)
        return;

    /* Ein Containerdurchlauf. Am Ende ist next == last und
       * prev == last-1 */
    while(next != last) {
        if (comp(*next, *prev))
            swap(*prev, *next);

        prev = next++;
    }
    mySort(first, prev, comp); // Rekursion mit [first, last-1)
}
```

Erlaubt nun auch ForwardIterator
und leere Container

std::sort

```
struct Quadrat {
    int px, py, len;
    Quadrat(int x, int y, int l):px(x), py(y), len(l) {}
};

bool operator<(const Quadrat& q1, const Quadrat& q2) {
    return q1.len < q2.len;
}

template<class T>
void print(const T& val) {
    cout << val << " ";
}

template<>
void print<Quadrat>(const Quadrat& q) {
    cout << "Quadrat mit Kantenlaenge " << q.len << ", ";
}

int main() {
    vector<Quadrat> q{Quadrat(0,0,5), Quadrat(5,5,10), Quadrat(10,10,2)};

    std::sort(q.begin(), q.end());
    for_each(q.begin(), q.end(), print<Quadrat>);
    cout << endl;
}
```

std::less verwendet operator<, wenn definiert

The diagram consists of a light green rectangular box containing the text 'std::less verwendet operator<, wenn definiert'. Two red arrows originate from this box. One arrow points diagonally upwards and to the left, ending at the 'operator<' definition in the code block above. The second arrow points diagonally downwards and to the left, ending at the 'std::sort' call in the 'main' function of the code block below.

std::for_each

```
#include <vector>
#include <iostream>

using namespace std;

template <class T>
void print(T val) {
    cout << val << " ";
}

int main() {
    vector<int> zahlen{9,8,7,6,5,4,3,2,1,0};

    for_each(zahlen.begin(), zahlen.end(), print);
}
```


Spezialisierte Templates

```
template <class T>
void print(const T &val) {
    cout << val << endl;
}
```

**generische Templatefunktion
für alle Typen T**

```
template <>
void print<Quadrat>(const Quadrat &q) {
    cout << "Quadrat mit Kantenlaenge " << q.len << endl;
}
```

Spezialisierung für Quadrat

```
struct Quadrat {
    int px; int py; int len;
    Quadrat(int x, int y, int l):px(x), py(y), len(l){};
};

int main() {
    vector<Quadrat> q{Quadrat(0,0,5), Quadrat(5,5,10), Quadrat(10,10,2)};

    mySort(q.begin(), q.end());
    print("Quadrats"); print(-123);
    for_each(q.begin(), q.end(), print<Quadrat>);
    print(Quadrat{9,12,3});
}
```


std::copy, std::copy_if

```
bool isEven(int i) {  
    return (i & 1) == 0;  
}
```

```
int main() {  
    vector<int> zahlen{9, 8, 7, 6, 5, 4, 3, 2, 1, 0};  
    vector<int> result;
```

**stellt sicher, dass Speicher in result
angelegt wird (result.reserve(...))**

```
    copy_if(zahlen.cbegin(), zahlen.cend(),  
            back_inserter(result),  
            isEven);  
    for_each(result.begin(), result.end(), print<int>);  
}
```

std::transform


Funktionsweise:

```
template<class Input, class Output, class Func>
Output transform(Input first1, Input last1, Output output, Func f) {
    while(first1 != last1) {
        *output++ = f(*first1++);
    }
    return output;
}
```

```
int main() {
    vector<int> zahlen{9,8,7,6,5,4,3,2,1,0};

    transform(zahlen.begin(), zahlen.end(), zahlen.begin(), mal2);
    for_each(zahlen.begin(), zahlen.end(), print);
}
```

**Ersetzt hier bestehende Elemente,
daher ausnahmsweise kein
insert_iterator notwendig.**



Wdh: Lambda-Ausdrücke

- `algorithm`-Funktionen sind ja ganz nützlich
 - Erprobte Algorithmen für oft benötigten Aufgaben wiederverwenden
 - Unabhängig vom Container-Typ
 - Oftmals kürzer zu schreiben als eigene `for`-Schleife
- Übergabe von Programm-Code (als Element-Operatoren oder Prädikate) mit externen Funktionen aber unnötig aufwendig
 - Wenn man dafür extra eine Funktionen schreiben muss...

```
int main() {  
    vector<int> zahlen{9,8,7,6,5,4,3,2,1,0};  
  
    transform(zahlen.begin(), zahlen.end(), zahlen.begin(), mal2);  
    for_each(zahlen.begin(), zahlen.end(), print);  
}
```

Lambda-Ausdrücke

- Funktionen als Werte auffassen
- Funktionen Variablen zuweisen (Funktionszeiger)
- Diese dann wie Zeiger auf normale Funktionen verwenden

```
int main() {  
    vector<int> zahlen{9,8,7,6,5,4,3,2,1,0};  
  
    auto mal2= [] (int val) -> int {return val*2;};  
  
    transform(zahlen.begin(), zahlen.end(), zahlen.begin(), mal2);  
    for_each(zahlen.begin(), zahlen.end(), print<decltype(zahlen)::value_type>);  
}
```

**Rückgabe-Datentyp spezifizieren.
Selten notwendig, da normalerweise
aus return ableitbar. (Hier redundant.)**

Lambda-Ausdrücke

- Umweg über Variablen nicht nötig
- Direkt als Funktionsparameter übergeben
- Bei `transform`, `for_each`, `accumulate` usw.

```
int main() {  
    vector<int> zahlen{9,8,7,6,5,4,3,2,1,0};  
  
    transform(zahlen.begin(), zahlen.end(), zahlen.begin(),  
              [] (int val) { return val*2; });  
    for_each(zahlen.begin(), zahlen.end(), [](int val) {  
                                                cout << val << " ";  
    });  
}
```

Lambda Capture

- Die meisten `algorithm`-Funktionen (wie `transform`) erwarten einstellige Funktionen
- Nur ein Parameter — das aktuelle Element...
- Mehrere Parameter nicht direkt möglich

Capture: angeben, wie lokale Variablen der Umgebung verwendet werden sollen. (Hier: alle Variablen by-reference)

```
7 int main() {  
8     vector<int> zahlen{9,8,7,6,5,4,3,2,1,0};  
9     int factor=2;  
10  
11     transform(zahlen.begin(), zahlen.end(), zahlen.begin(), [&] (int val) {return val*factor;});  
12     for_each(zahlen.begin(), zahlen.end(), [] (int val) {cout << val << " ";});  
13 }
```

https://en.cppreference.com/w/cpp/language/lambda#Lambda_capture

Lambda Capture

- Die meisten `algorithm`-Funktionen (wie `transform`) erwarten einstellige Funktionen
- Nur ein Parameter — das aktuelle Element...
- Mehrere Parameter nicht direkt möglich

Capture
(hier: by-value mit Initialisierung)



```
int main() {  
    vector<double> q{17.2, 9.0, 13.123, 77.5};  
  
    double avg = accumulate(q.begin(), q.end(), 0.0,  
                            [size = q.size()] (double init, double value) {  
                                return init + value/size;  
                            });  
    cout << "Der Durchschnitt betraegt " << avg << endl;  
}
```

https://en.cppreference.com/w/cpp/language/lambda#Lambda_capture

Zufallszahlen mit `mt19937`

- Zufallszahlenfolge basierend auf Mersenne-Primzahl $2^{19937} - 1$
- Höhere Qualität als `rand()`
- Optionale Verteilungsfunktionen (uniform, Normalverteilung, exponentiell)

```
#include <iostream>
#include <random>

using namespace std;

int main() {
    vector<int> zahlen;

    generate_n(back_inserter(zahlen), 100, mt19937{});
    for_each(zahlen.begin(), zahlen.end(), print<int>);
}
```

Zufallszahlen mit mt19937

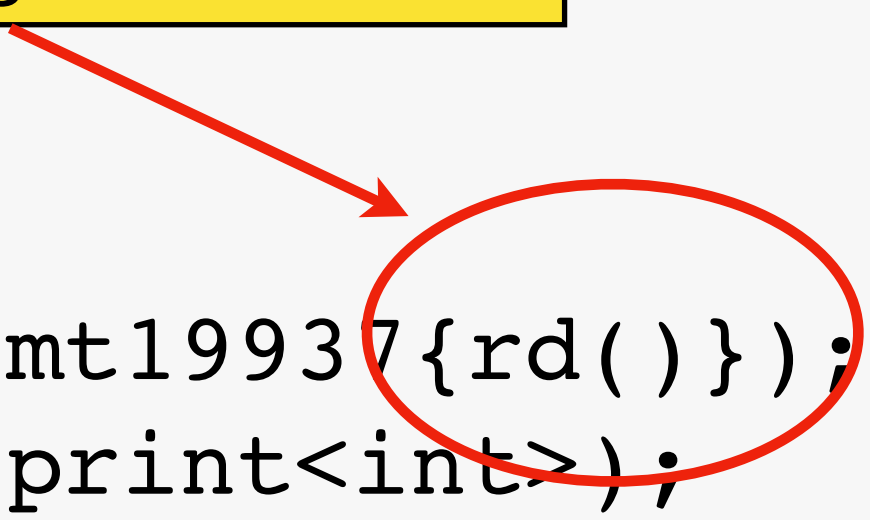
```
#include <iostream>
#include <random>

using namespace std;

int main() {
    vector<int> zahlen;
    random_device rd;

    generate_n(back_inserter(zahlen), 100, mt19937{rd()});
    for_each(zahlen.begin(), zahlen.end(), print<int>);
}
```

Wichtig!
Initialisieren mit seed für echte Zufallszahlenfolge.



std::bind: Argumente binden

- Erzeugt Funktor
 - Argumente können festgelegt werden
 - Platzhalter `_1`, `_2`, `_3` ... für nicht gebundene Argumente

```
#include <algorithm>
#include <iostream>
#include <functional>
#include <vector>
```

```
using namespace std;
using namespace std::placeholders;
```

```
int mult(int a, int b) {
    return a * b;
}
```

```
int main() {
    vector<int> zahlen{9,8,7,6,5,4,3,2,1,0};
```

```
    transform(zahlen.begin(), zahlen.end(), zahlen.begin(), bind(mult, 5, _1);
    for_each(zahlen.begin(), zahlen.end(), print);
}
```

Binäre Funktion,
`bind` erzeugt unären Funktor,
Erstes Argument festgelegt (= 5)

Platzhalter für Werte von `zahlen`

std::accumulate

accumulate()
reduce()

```
#include <iostream>
#include <numeric>
#include <vector>

using namespace std;

int mult(int a, int b) {
    return a * b;
}

int main() {
    vector<int> zahlen{9,8,7,6,5,4,3,2,1};

    int result = accumulate(zahlen.begin(), zahlen.end(), 1, mult);
    cout << result << endl;
}
```

Binäre Funktion

Initialwert für Parameter init

Stream-Iteratoren (1/2)

- Auf I/O-Streams über Iterator-Schnittstelle zugreifen
- Template-Argument ist Element-Typ
 - d.h., Typ der Werte, die gelesen werden sollen
 - und die der Iterator liefern soll
- Vorteile?

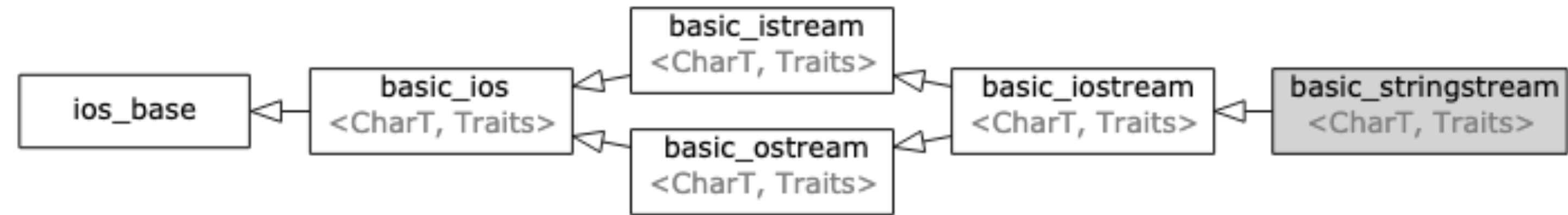
```
1 #include <fstream>
2 #include <string>
3 #include <iostream>
4
5 using namespace std;
6
7 int main() {
8     ifstream datei("stream-it.cpp");
9     istream_iterator<string> wort(datei);
10    istream_iterator<string> ende;
11
12    while (wort != ende) {
13        cout << *wort++ << endl;
14    }
15 }
```


Stream-Iteratoren (2/2)

- Funktioniert auch mit Ausgabe-Iteratoren
- `ostream_iterator`
- Man kann dabei noch Trennzeichen als Parameter angeben

```
1 #include <fstream>
2 #include <string>
3 #include <iostream>
4
5 using namespace std;
6
7 int main() {
8     ifstream datei("stream-it.cpp");
9     istream_iterator<string> wort(datei);
10    istream_iterator<string> ende;
11    ofstream ziel("Ergebnis.txt");
12    ostream_iterator<string> ausgabe(ziel, "*\n");
13
14    while(wort != ende) {
15        *ausgabe++ = *wort++;
16    }
17 }
```

std::stringstream



```
1 #include <sstream>
2 #include <iostream>
3
4 using namespace std;
5
6 int main() {
7     stringstream s;
8
9     s << "Hallo, hallo." << endl;
10
11     cout << s.str();
12
13     return 0;
14 }
```

```
1 #include <sstream>
2 #include <iostream>
3
4 using namespace std;
5
6 int main() {
7     stringstream s("Von einem string "
8                   "wie aus einem istream lesen");
9     string res;
10
11     s >> res;
12     cout << res << endl;
13
14     return 0;
15 }
```

Textzeilen mit stringstream parsieren

```
1 #include <sstream>
2 #include <iostream>
3
4 using namespace std;
5
6 int main() {
7     string input("Diesen String in Wörter zerlegen");
8
9     stringstream tokenStream(input);
10    string token;
11
12    while (getline(tokenStream, token, ' ')) {
13        cout << token << endl;
14    }
15    return 0;
16 }
```

```
bash-3.2$ ./parse_line
Diesen
String
in
Wörter
zerlegen
bash-3.2$ █
```

Textzeilen mit stringstream parsieren – als Funktion

```
1 #include <sstream>
2 #include <iostream>
3
4 using namespace std;
5
6 ostream& split(const string& input, ostream& out) {
7     istringstream tokenStream(input);
8     string token;
9
10    while (getline(tokenStream, token, ' ')) {
11        out << token << endl;
12    }
13    return out;
14 }
15
16 int main() {
17     split("Diesen String in Wörter zerlegen", cout) << endl;
18     split("Diesen String auch...", cout) << endl;
19
20     return 0;
21 }
```

```
bash-3.2$ ./split1
Diesen
String
in
Wörter
zerlegen

Diesen
String
auch...

bash-3.2$ █
```

Textzeilen mit stringstream parsieren – als Funktion

Ergebnis in vector...

```
1 #include <sstream>
2 #include <iostream>
3 #include <vector>
4
5 using namespace std;
6
7 typedef vector<string> stringvec;
8
9 stringvec& split(const string& input, stringvec& out) {
10     stringstream tokenStream(input);
11     string token;
12
13     while (getline(tokenStream, token, ' ')) {
14         out.push_back(token);
15     }
16     return out;
17 }
18
19 void println(const string& s) {
20     cout << s << endl;
21 }
22
23 int main() {
24     stringvec res;
25
26     split("Diesen String in Wörter zerlegen", res);
27
28     for_each(res.begin(), res.end(), println);
29
30     return 0;
31 }
```

```
bash-3.2$ ./split2
Diesen
String
in
Wörter
zerlegen
bash-3.2$ █
```

Textzeilen mit `stringstream` parsieren – generisch?

- Eine `split`-Funktion für unterschiedliche Anwendungen
 - In `ostreams` und Container schreiben
 - Benutzerdefinierte Trennzeichen

Textzeilen mit stringstream parsieren – generisch

```
1 #include <sstream>
2 #include <iostream>
3 #include <vector>
4 #include <iterator>
5
6 using namespace std;
7
8 typedef vector<string> stringvec;
9
10 template<class OutputIterator>
11 OutputIterator split(const string& s, OutputIterator it, char delimiter=' ')
12 {
13     string token;
14     istringstream tokenStream(s);
15     while (getline(tokenStream, token, delimiter))
16     {
17         *it++=token;
18     }
19     return it;
20 }
21
22
23 void println(const string& s) {
24     cout << s << endl;
25 }
26
27 int main() {
28     stringvec res;
29
30     split("Diesen String in Wörter zerlegen", ostream_iterator<string>(cout, "\n"));
31     cout << endl;
32
33     split("Diesen String auch", back_inserter(res));
34     for_each(res.begin(), res.end(), println);
35
36     return 0;
37 }
```

```
bash-3.2$ ./split3
Diesen
String
in
Wörter
zerlegen

Diesen
String
auch
bash-3.2$ █
```

https://www.cplusplus.com/reference/iterator/ostream_iterator/

Übung

- Schreiben Sie ein Programm, das einen beliebig langen Text aus einer Datei lesen und das Vorkommen unterschiedlicher Wörter zählen kann. Das Programm soll nach dem Einlesen aller Wörter eine Tabelle ausgeben. Die Tabelle soll nach den Häufigkeiten sortiert sein. Gross- und Kleinschreibung soll nicht relevant sein.
- Beispiel
 - *Jeder hat das Recht, seine Meinung in Wort, Schrift und Bild frei zu äußern und zu verbreiten und sich aus allgemein zugänglichen Quellen ungehindert zu unterrichten. Die Pressefreiheit und die Freiheit der Berichterstattung durch Rundfunk und Film werden gewährleistet. Eine Zensur findet nicht statt.*
 - und: 4
 - zu: 3
 - die: 2
 - das: 1
 - usw...