Teil 10
Standardbibliothek

- Überblick
- sortieren
- binäre Suche
- Sequenzoperationen
- Tupel

Überblick

- string
- istream / ostream
- array / vector
- list
- map / unordered_map / set
- queue / stack
- tuple
- shared_ptr / unique_ptr 

- complex
- exception
- thread / mutex 

- regexp
- ...

- fabs, fmod, exp, log, pow, sqrt, sin, cos, ceil, floor...
- min / max
- swap
- sort
- for_each
- ...

sortieren

```
#include <algorithm>

bool greater(int a, int b)
{
  return a > b;
}

std::vector<int> list = {46, 44, 62, 12, 37};
std::sort(list.begin(), list.end());

std::sort(list.begin(), list.end(), greater);
```

- normalerweise schneller aus qsort (Relikt aus C)
- optional mit Vergleichsfunktion oder -funktor

sortieren

```
#include <algorithm>

struct Greater
{
  bool operator()(int a, int b)
  {
    return a > b;
  }
}

std::vector<int> list = {46, 44, 62, 12, 37};
Greater greater;
std::sort(list.begin(), list.end(), greater);
```

- Funktoren sind Objekte mit Klammer-Operator
- im Gegensatz zu Funktionen nicht zustandslos

sortieren

```
#include <algorithm>

std::vector<int> list = {46, 44, 62, 12, 37};
std::sort(list.begin(), list.end(),
          [](int a, int b) -> bool { return a > b });
```

- seit C++11 kann man auch „Lambdas“ nutzen, d.h. namenlose Funktionen, die sich auch direkt beim Aufruf definieren lassen

```
#include <algorithm>

std::vector<int> list = {46, 44, 62, 12, 37};
auto greater = [](int a, int b) -> bool { return a > b };
std::sort(list.begin(), list.end(), greater);
```

binäre Suche

```
#include <algorithm>

std::vector<int> list = {46, 44, 62, 12, 37};
std::sort(list.begin(), list.end());
auto iterator = std::lower_bound(list.begin(), list.end(), 40);
if (iterator != list.end())
{
  cout << (iterator - list.begin()) << endl; // 2
  cout << *iterator << endl; // 44
}
```

- nur für sortierte Folgen

Sequenzoperationen

```
#include <algorithm>

struct Summation
{
  int sum = 0;
  bool operator()(const int n)
  {
    sum += n;
  }
}

std::vector<int> list = {46, 44, 62, 12, 37};
Summation summation;
std::for_each(list.begin(), list.end(), summation);
cout << summation.sum;
```

- anstatt Schleifen
- Vorteil: Operation leicht austauschbar

Sequenzoperationen

```
#include <algorithm>

std::vector<int> list = {46, 44, 62, 12, 37};

int sum = 0;
auto summation = [&sum](const int n) { sum += n; };

std::for_each(list.begin(), list.end(), summation);
cout << sum;
```

- ab C++11 auch mit Lambdas möglich

Tupel

C++ 11

```
#include <tuple>

std::tuple<int, std::string> getError()
{
  return std::make_tuple(404, "not found");
}

auto error = getError();
auto errorCode = std::get<0>(error);
auto errorText = std::get<1>(error);
```

```
std::tuple<int, std::string> getError()
{
  return {404, "not found"};
}
```

C++ 17

...nachschlagen...

- Webseiten:
  http://www.cplusplus.com/
  http://en.cppreference.com/
  http://de.cppreference.com/