Generizität mit Templates

Vorkurs C/C++, Olaf Bergmann

Namensräume

- Namenskonflikte vermeiden über `namespace`
  - Schachtelbar
  - Benutzung mit Scope-Operator `::`
- `std`: Namensraum für Definitionen aus der Standard-Bibliothek
- Leerer Bezeichner für oberste Ebene

```
namespace X {
  namespace Y {
    int i;
  }
}

X::Y::i = 7;
```

```
bool
Obj::read(const std::string &f) {
  ::read(file...);
}
```

Argument-dependent Lookup

- Überladene Funktionen werden im Namespace der Argumente gesucht:

```
std::cout << 5;
```

Kein operator << im globalen Namespace, aber definiert für std::cout.

- Idiom:

```
using std::swap;
swap(a, b);
```

Verwendet std::swap, falls kein spezifisches swap auf a oder b definiert

Templates

- Klasse oder Funktion mit Typ-Parameter

```
template <class T>
class vector {
public:
   ...
   int size() const;
private:
   int sz;
   T *p;
};

template<class T>
int vector<T>::size() const { return sz; }
```

Benutzung:

```
vector<int> vi;
vector<char> vc;
```

Typinferenz: decltype

Wdh.:

```
typedef vector<int> Zahlen;

Zahlen zahlen = { /* ... */ }

for (Zahlen::const_iterator i = zahlen.begin(); i != zahlen.end(); ++i) {
  Zahlen::value_type z = *i;
  /* ... */
}
```

viel zu schreiben,
sehr unübersichtlich

```
typedef vector<int> Zahlen;

Zahlen zahlen = { /* ... */ }

for (Zahlen::co                    = za
  decltype(*i) z = *i;
  /* ... */
}
```

... aber immer noch
viel zu schreiben ...

decltype(*expr*):
bestimmt Typ von *expr* zur Übersetzungszeit

```
decltype(2 + 7)      → int

Point *p = nullptr;
decltype(*p)         → Point&

Typ::iterator i;
decltype(*i)         → Typ::value_type&
```

Typinferenz: auto

Wdh.:

```
typedef vector<int> Zahlen;

Zahlen zahlen = { /* ... */ }

for (Zahlen::const_iterator i = zahlen.begin(); i != zahlen.end(); ++i) {
  Zahlen::value_type z = *i;
  /* ... */
}
```

Zahlen::value_type
ergibt sich aus *i

```
typedef vector<int> Zahlen;

Zahlen zahlen = { /* ... */ }

for (Zahlen::const_iterator i =
  auto z = *i;
  /* ... */
}
```

auto:
Typ zur zur Übersetzungszeit ermitteln

```
int x, a[10];
auto y = x;        → int

auto p = a;        → int *
auto *q = &a[0];   → int *
auto &r = a[0];    → int &

Typ::iterator i;
auto v = *i;       → Typ::value_type &
```

Typinferenz: auto

Wdh.:

```
typedef vector<int> Zahlen;

Zahlen zahlen = { /* ... */ }

for (Zahlen::const_iterator i = zahlen.begin(); i != zahlen.end(); ++i) {
  auto z = *i;
  /* ... */
}
```

int &

```
typedef vector<int> Zahlen;

Zahlen zahlen = { /* ... */ }

for (int& z : zahlen) {

  /* ... */
}
```

```
typedef vector<int> Zahlen;

Zahlen zahlen = { /* ... */ }

for (auto& z : zahlen) {

  /* ... */
}
```

Typinferenz: auto als Rückgabetyp

```
auto fib(unsigned long i)
{
   switch (i) {
   case 0:
   case 1:
      return i;
   default:
      return fib(i - 2) + fib(i - 1);
   }
}
```

Bestimmt Rückgabetyp

Rekursion erlaubt, wenn Rückgabetyp bekannt

Typinferenz: auto als Rückgabetyp

```
auto fib(unsigned long i)  -> decltype(i)
{
   switch (i) {
   default:
     return fib(i - 2) + fib(i - 1);
   case 0:
   case 1:
     return i;
   }
}
```

benötigt, wenn Rückgabetyp nicht automatisch bestimmbar

Fehler: Compiler kennt Rückgabetyp noch nicht

Algorithmen

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

double
average(const std::vector<double> &v) {
  return std::accumulate(v.cbegin(), v.cend(), 0.0) / v.size();
}
```

Mit Lambda-Ausdruck:

erlaubt Variablen-Referenzen

```
std::accumulate(v.cbegin(), v.cend(), 0.0,
     [&] (double init, double val) {
         return init + val / v.size();
     });
}
```

Algorithmen und Iteratoren

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

using namespace std;

vector<int> v;
generate_n(inserter(v, v.end()), 100, mt19937());
```

erzeugt Werte für Container

Pseudo-Zufallszahlengenerator

erzeugt einen Insert-Iterator für v

```
#include <iostream>
#include <iterator>

using namespace std;

copy(v.begin(), v.end(), ostream_iterator<int>(cout, ", "));
```

kopiert Elemente von [begin(), end())

Ziel der Kopie: Ausgabeiterator

Generator

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

using namespace std;

class Data {
  int wert = 0;
public:
  int operator()(void) { return wert++; }
};

vector<int> v;
generate_n(inserter(v, v.end()), 100, Data());
```

spart den Konstruktor für die Initialisierung

Funktionsaufrufoperator, z. B.: Data()();