Typsynonyme mit `using`

- Eigene Namen für bereits existente Typen
- `using` t' `=` t`;`

```
using size_t = unsigned int;
```

```
using myvector = double[10];
```

```
using mypointer = double*;
```

Klassen-assoziierte Typsynonyme

- Definition von Typsynonym innerhalb von Klasse möglich
- Steht innerhalb von Klasse unqualifiziert zur Verfügung
- Außerhalb von Klasse K assoziiertes Typsynonym mit Name t mit K`::`t

Assoziierte Typsynonyme für Klasse `Vektor`

vector_types.cpp

```cpp
#include <iostream>

using namespace std;

class Vektor {
  public:
    using size_type = unsigned int;

  private:
    double* ap;
    size_type len;

  public:
    Vektor(size_type n = 0, double x = 0) : len(n) {
      ap = new double[n];
      for (size_type i = 0; i < n; i++) ap[i] = x;
    }
    ~Vektor() { delete[] ap; }

    double& operator[](size_type i) {
      return ap[i];
    }
    size_type size() const {
      return len;
    }
};

int main() {
  Vektor v{4};
  for (Vektor::size_type i = 0; i < v.size(); i++)
    v[i] = i;

  cout << "v: ";
  for (Vektor::size_type i = 0; i < v.size(); i++)
    cout << v[i] << " ";
  cout << endl;

  return 0;
}
```

```
v: 0 1 2 3
```

Typsynonyme mit `typedef`

- In neuem Code eher mit `using`
- `typedef` t t'; $\leftrightarrow$ `using` t' = t;

```cpp
typedef unsigned int size_t;
```

```cpp
typedef double myvector[10];
```

```cpp
typedef double *mypointer;
```

Iteratoren allgemein

- Iterator ist Wert von Typ assoziiert mit Behälter
- Modelliert Position eines Elements im Behälter
- Immer `operator*` geeignet (oft überladen) sodass Rückgabewert Element liefert oder Referenz darauf
- Oft `operator++` bzw. `operator--` für nachste bzw. vorherige Position
- I.d.R. kein Konstruktor sondern Rückgabewert von Behälter-Methoden

Iteratoren für Klasse `Vektor`

vector_iterator.cpp

```cpp
#include <iostream>

using namespace std;

class Vektor {
  private:
    double* ap;
    unsigned int len;

  public:
    Vektor(int n = 0, double x = 0) : len(n) {
      ap = new double[n];
      for (int i = 0; i < n; i++) ap[i] = x;
    }
    ~Vektor() { delete[] ap; }

    class iterator {
      friend class Vektor;

      private:
        Vektor* v;
        unsigned int pos;

        iterator(Vektor* v_, unsigned int pos_ = 0)
          : v(v_), pos(pos_) {}

      public:
        double& operator*() {
          return v->ap[pos];
        }
        iterator& operator++() {
          if (pos < v->len) pos++;
          return *this;
        }
        bool operator!=(const iterator& other) const {
          return v != other.v || pos != other.pos;
        }
    };

    iterator begin() { return iterator{this}; }
    iterator end() { return iterator{this, len}; }
};

int main() {
  Vektor v{4};

  cout << "v: ";
  unsigned int count = 0;
  for (Vektor::iterator it = v.begin(); it != v.end(); ++it) {
    *it = count++;
    cout << *it << " ";
  }
  cout << endl;
}
```

```
v: 0 1 2 3
```

Iteratoren über konstante Behälter

- I.d.R. separater Typ für Iteratoren über Konstanten (analog zu `const Vektor* v` statt `Vektor* v`)
- Separate `const`-Methoden auf Behälter um Konstanteniteratoren zu erstellen
- Analog zu Zugriffsschutz durch pointer auf Konstante:
  `double arr[1]; const double* arrp = arr; arrp[0] = 17;`

Konstanteniteratoren für Klasse `Vektor`

vector_const_iterator.cpp

```cpp
#include <iostream>

using namespace std;

class Vektor {
  private:
    double* ap;
    unsigned int len;

  public:
    Vektor(int n = 0, double x = 0) : len(n) {
      ap = new double[n];
      for (int i = 0; i < n; i++) ap[i] = x;
    }
    ~Vektor() { delete[] ap; }

    class const_iterator {
      friend class Vektor;

      private:
        const Vektor* v;
        unsigned int pos;

        const_iterator(const Vektor* v_, unsigned int pos_ = 0)
          : v(v_), pos(pos_) {}

      public:
        const double& operator*() const
          { return v->ap[pos]; }
        const_iterator& operator++() {
          if (pos < v->len) pos++;
          return *this;
        }
        bool operator!=(const const_iterator& other) const
          { return v != other.v || pos != other.pos; }
    };

    const_iterator begin() const
      { return const_iterator{this}; }
    const_iterator end() const
      { return const_iterator{this, len}; }
};

int main() {
  const Vektor v{2, 1.141};

  cout << "v: ";
  for ( Vektor::const_iterator it = v.begin();
        it != v.end();
        ++it )
    cout << *it << " ";
  cout << endl;
}
```

```
v: 1.141 1.141
```

Iteratoren für STL `vector`

- Mit `vector<T> v`:
  - `vector<T>::iterator vi1 = v.begin(), vi2 = v.end();`
  - `vector<T>::const_iterator vi1 = v.begin(), vi2 = v.end();`
  - `vector<T>::reverse_iterator vi1 = v.rbegin(), vi2 = v.rend();`
  - `vector<T>::const_reverse_iterator vi1 = v.rbegin(), vi2 = v.rend();`
- Iteratoren für `vector<T>` sind *random-access iterators*, d.h. mit `a` und `b` Iteratoren, `n` ganzzahling, `m` Komponente von `T`, `t` vom Typ `T`:
  - `a == b`, `a != b`
  - `a < b`, `a <= b`, `a > b`, `a >= b`
  - `*a` und `a->m`
  - `*a = t`
  - `a++`, `++a`, `--a`, `a--`
  - `b - a`
  - `a + n`, `n + a`, `a += n`, `a - n`, `a -= n`
  - `a[n]`

Matrixmultiplikation mit Iteratoren

matmul.cpp

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

using namespace std;

int main() {
  unsigned int m, n;
  cout << "m, n: "; cin >> m >> n;

  vector<double> b(n);
  cout << "b: ";
  for (vector<double>::iterator i = b.begin();
    i != b.end();
    ++i)
    cin >> *i;

  vector<vector<double>> a(m, vector<double>(n));
  for (unsigned int i = 0; i < m; i++) {
    cout << "a[" << i << "][...]: ";
    for (unsigned int j = 0; j < n; j++)
      cin >> a[i][j];
  }

  vector<double> c(m);
  vector<double>::iterator cpos = c.begin();

  for (vector<vector<double>>::iterator apos_i = a.begin();
    apos_i != a.end();
    apos_i++, cpos++) {
    vector<double>::iterator bpos = b.begin();
    for (vector<double>::iterator apos_j = (*apos_i).begin();
      apos_j != (*apos_i).end();
      apos_j++, bpos++)
      *cpos += *apos_j * *bpos;
  }

  cout << "a*b: ";
  for (cpos = c.begin(); cpos != c.end(); cpos++)
    cout << *cpos << " ";
  return 0;
}
```

```
m, n: 2 3
b: 1 2 3
a[0][...]: 4 5 6
a[1][...]: 7 8 9
a*b: 32 50
```

Einschub: Typ-Platzhalter

- Deklaration und Initialisierung von Variable mit `auto` statt Typ
- Typ wird anhand der Initialisierung automatisch bestimmt

```
int main()
{
  auto i = 1;       // i: int
  const auto k = 5; // k: const int
  auto x = 1.0;     // x: double

  auto& j = i; // j: int&
  auto& l = k; // l: const int&

  const auto& n = 2; // n: const int&

  auto p = 2, y = 2.0; // unzulaessig
}
```

Range-based for loops

- Spezielle Syntax für `for`-Schleifen mit Iteratoren
- `T` oft `auto` oder `auto&`

Range-based for syntax

```
for (T i : v) { ... }
```

Äquivalentes konventionelles for

```
for (auto __pos = v.begin(),
          __end = v.end();
     __pos != __end;
     ++__pos) {
  T i = *__pos;
  ...
}
```

Matrixmultiplikation mit Iteratoren II

matmul_range.cpp

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

using namespace std;

int main() {
  unsigned int m, n;
  cout << "m, n: "; cin >> m >> n;

  vector<double> b(n);
  cout << "b: ";
  for (auto& i: b)
    cin >> i;

  vector<vector<double>> a(m, vector<double>(n));
  for (unsigned int i = 0; i < m; i++) {
    cout << "a[" << i << "][...]: ";
    for (auto& j: a[i])
      cin >> j;
  }

  vector<double> c(m);
  vector<double>::iterator cpos = c.begin();

  for (auto i: a) {
    vector<double>::const_iterator bpos = b.begin();
    for (auto j: i)
      *cpos += j * *(bpos++);
    cpos++;
  }

  cout << "a*b: ";
  for (auto i: c)
    cout << i << " ";
  return 0;
}
```

```
m, n: 2 3
b: 1 2 3
a[0][...]: 4 5 6
a[1][...]: 7 8 9
a*b: 32 50
```

Verwendung von Konstanteniteratoren

vector_norm.cpp

```cpp
#include <vector>
#include <iostream>
#include <cmath>

using namespace std;

double Norm(const vector<double>& x) {
  double s = 0;
  for (const double& c: x)
    s += c*c;
  return sqrt(s);
}

int main() {
  vector<double> a{5, 1};
  cout << "Norm(a) = " << Norm(a) << endl;
  return 0;
}
```

```
Norm(a) = 5.09902
```