## Typsynonyme

⟨Typdefinition⟩ ⟶ "`using`" ⟨Name⟩ "`=`" ⟨Typ⟩ "`;`"

Mit dem Schlüsselwort `using` können neue Namen für bereits existente Typen vereinbart werden.

*Beispiel* (Einfache Typsynonyme).

```
using size_t = unsigned int;
using myvector = double[10];
using mypointer = double*;
```

Die Definition von Typsynonymen ist auch innerhalb von Klassen möglich. Zugriff auf so definierte Namen folgt den selben Regeln wie der Zugriff auf Namen von Klassenkomponenten. D.h. innerhalb der Klasse steht der Name unqualifiziert zur Verfügung, außerhalb davon qualifiziert mit dem Namen der Klasse. Für eine Typsynonym $t$ innerhalb einer Klasse $K$ also als $K::t$.

Zugriff auf Typsynonyme folgt den Sichtbarkeitseinschränkungen `private` bzw. `public`, wie gewöhnlich.

*Beispiel.* Wir definieren ein assoziiertes Typsynonym für Längen von Vektoren einer rudimentären eigenen Klasse `Vektor`.

vector_types.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

⟨Typdefinition⟩ → "`typedef`" ⟨Typ⟩ ⟨Deklaration⟩ "`;`"

Typsynonyme können auch mit einer älteren Schreibweise mit `typedef` definiert werden. Die Definition mit `using` ist in aller Regel vorzuziehen.

*Beispiel* (Einfache Typsynonyme).

```
typedef unsigned int size_t;
typedef double myvector[10];
typedef double *mypointer;
```

## Iteratoren

Iteratoren sind mit einer Behälterklasse assoziierte Werte, die eine Position in einem konkreten Objekt der Behälterklasse modelliert.

Valide Iteratoren können mit `operator*` dereferenziert werden um Zugriff auf den aktuellen Wert an der referenzierten Position des assoziierten Behälters zu erhalten. Iteratoren können invalidiert werden, wenn der assoziierte Behälter verändert wird. In diesem Fall gibt es keine Garantien bzgl. dem Verhalten von `operator*`.

### Iteratoren und weitere Funktionen für vector

Im Folgenden Bezeichnet $v$ einen Vektor vom Typ `vector<`$T$`>`, $n$ eine ganze Zahl, $c$ eine Klassenkomponente der Klasse $T$, $t$ ein Wert vom Typ $T$, $i$, $i'$ Iteratoren über $v$ und $i_1$, $i_2$ Iteratoren über einen anderen Vektor.

| | |
|---|---|
| `vector<`$T$`>::iterator` $i$ | Vereinbart Variable $i$ als Vorwärts-Iterator |
| `vector<`$T$`>::const_iterator` $i$ | Vereinbart Variable $i$ als Vorwärts-Konstanteniterator |
| `vector<`$T$`>::reverse_iterator` $i$ | Vereinbart Variable $i$ als Rückwärts-Iterator |

| | |
|---|---|
| `vector<`$T$`>::const_reverse_iterator` $i$ | Vereinbart Variable $i$ als Rückwärts-Konstanteniterator |
| $v$`.begin()` | Liefert Vorwärts-Iterator über $v$ zum Index `0` |
| $v$`.end()` | Liefert Vorwärts-Iterator zum Index $v$`.size()`, eine Position *nach Ende* des Vektors |
| $v$`.rbegin()` | Liefert Rückwärts-Iterator über $v$ zum Index $v$`.size() - 1` |
| $v$`.rend()` | Liefert Rückwärts-Iterator zum Index `-1`, eine Position *vor Beginn* des Vektors |
| `++`$i$ $i$`++` | Verschiebt Iterator $i$ auf nächste Position in Durchlaufrichtung |
| `--`$i$ $i$`--` | Verschiebt Iterator $i$ auf vorherige Position in Durchlaufrichtung |
| $i$ `+` $n$ $n$ `+` $i$ $i$ `-` $n$ | Liefert Iterator $i$ verschoben um $n$ in Durchlaufrichtung |
| $i$ `+=` $n$ $i$ `-=` $n$ | Verschiebt Iterator $i$ um $n$ in Durchlaufrichtung |
| `*`$i$ | Referenz zu Wert an aktueller Position von $i$ |
| $i$`[`$n$`]` | `*(`$i$ `+` $n$`)` |
| $i$`->`$c$ | `*(`$i$`).`$c$ |
| $v$`.insert(`$i$`,` $t$`)` | Fügt vor aktueller Position von $i$ Kopie von $t$ als neues Element in $v$ ein. Liefert Iterator auf Position des eingefügten Elements zurück. |
| $v$`.insert(`$i$`,` $i_1$`,` $i_2$`)` | Fügt vor aktueller Position von $i$ Kopien der Elemente im Interval $[i_1, i_2)$ in $v$ ein. Liefert Iterator auf Position des ersten eingefügten Elements zurück. |
| $v$`.erase(`$i$`)` | Löscht Element an aktueller Position von $i$ aus $v$. Liefert Iterator auf Element das direkt auf nun gelöschtes Element gefolgt hat. |
| $v$`.erase(`$i$`,` $i'$`)` | Löscht Elemente im Intervall $[i, i')$ aus $v$. Liefert Iterator auf Element das direkt auf letztes nun gelöschtes Element gefolgt hat. |

*Beispiel.* Wir definieren ein Hauptprogramm um eine Matrix und einen Vektor von der Standardeingabe einzulesen, miteinander zu multplizieren und den Ergebnis-Vektor auszugeben.

matmul_w.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
```

*Beispiel.* Wir definieren eine Funktion um die euklidische Norm eines Vektors zu berechnen und ein einfaches Hauptprogramm um die Funktion zu testen.

vector_norm_expl.cpp

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

using namespace std;

double Norm(const vector<double>& x) {
  double s = 0;
  for (vector<double>::const_iterator c = x.begin(); c != x.end(); ++c)
    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
```

## Typ-Platzhalter

Das Schlüsselwort `auto` kann bei Deklaration von Variablen anstelle eines Typnamens verwendet werden und gibt an, dass anhand der Initialisierung der Variable automatisch eine passende Belegung an Stelle von `auto` abgeleitet werden soll.

Werden mehrere Variablen auf einmal vereinbart, so muss die abgeleitete Belegung von `auto` für alle Variablen identisch sein.

*Beispiel.* Wir vereinbaren einige Variablen unter Verwendung von `auto`.

```
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
}
```

## Bereichsbasierte `for`-Schleifen

⟨RangeFor⟩ → “`for`” “`(`” ⟨VarDekl⟩ “`:`” ⟨Ausdruck⟩ “`)`” ⟨Anweisung⟩

Die folgenden Anweisungen sind i.W. äquivalent:

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

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

*Beispiel.* Wir definieren unter Verwendung bereichsbasierter `for`-Schleifen ein Hauptprogramm um eine Matrix und einen Vektor von der Standardeingabe einzulesen, miteinander zu multplizieren und den Ergebnis-Vektor auszugeben.

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
```