Iteratoren und Listenfunktionen in der Behälterklasse vector

Der `vector`-Datentyp stellt vier Iteratoren mit wahlfreiem Zugriff (random access iterator) bereit. Für jede Durchlaufrichtung gibt es zwei Iteratoren, davon jeweils einen für Verweise auf konstante Komponenten. Iteratoren können bei einem resize ungültig werden.

Operation	*Wirkung*
`vector<T>::iterator pos`	vereinbart Vorwärtsiteratorvariable *pos*
`vector<T>::const_iterator pos`	vereinbart Konstantenvorwärtsiteratorvariable *pos*
`vector<T>::reverse_iterator pos`	vereinbart Rückwärtsiteratorvariable *pos*
`vector<T>::const_reverse_iterator pos`	vereinbart Konstantenrückwärtsiteratorvariable *pos*
`a.begin()`	liefert Vorwärtsiterator für die Komponente zum Index 0
`a.end()`	liefert Vorwärtsiterator für die Komponente zum Index `a.size()` (Position *nach* der letzten Komponente!)
`a.rbegin()`	liefert Rückwärtsiterator für die letzte Komponente
`a.rend()`	liefert Rückwärtsiterator für die Position *vor* der ersten Komponente, d.h *vor* der Komponente zum Index 0
`++pos pos++`	inkrementiert Iterator *pos*
`--pos pos--`	dekrementiert Iterator *pos*
`pos+i i+pos pos-i`	*i*-te Position nach bzw. vor *pos*
`pos+=i pos-=i`	Iterator *pos* um *i* Positionen vor- bzw. zurücksetzen
`*pos`	Komponentenwert zur Iteratorposition *pos*
`pos[i]`	`*(pos+i)`
`pos->comp`	`(*pos).comp`
`a.insert(pos)`	fügt an der Position *pos* eine neue nach Voreinstellung initialisierte Komponente ein; liefert Position der neuen Komponente
`a.insert(pos,t)`	fügt an der Position *pos* eine neue mit dem Wert *t* ein initialisierte Komponente ein
`a.insert(pos,anf,end)`	fügt an der Position *pos* Kopien der Komponenten im Iteratorbereich $[anf, end)$ ein
`a.erase(pos)`	löscht Komponente an der Position *pos*; liefert Position der folgenden Komponente
`a.erase(anf,end)`	löscht Komponenten, auf die der Iteratorbereich $[anf, end)$ verweist; liefert Position der folgenden Komponente

Beispiel: $a \in \mathbb{R}^{m\times n}$, $b \in \mathbb{R}^n$, $\mathbb{R}^m \ni c = Ab$

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

using namespace std;

int main()                  // Berechne c = A*b   A in R^(m x n), b in R^n, c in R^m
{
  int m,n;
  cout << "m n: "; cin >> m >> n;
  vector<double> b(n),c(m);
  vector<vector<double>> a(m,vector<double>(n));

  cout << "Vektor b: " << endl;
  for (vector<double>::iterator bpos=b.begin(); bpos!=b.end(); ++bpos)
    cin >> *bpos;
```

```
  cout << "Matrix a: " << endl;
  for (vector<vector<double>>::iterator apos_i=a.begin(); apos_i!= a.end(); ++apos_i) {
    for (vector<double>::iterator apos_j=apos_i->begin(); apos_j!=apos_i->end(); ++apos_j)
      cin >> *apos_j;
  }

  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 << "  ";

  cout << endl << "a.size()="<< a.size() << " b.size()=" << b.size() << endl;
  return 0;
}
```

Ausgabe:

```
m n: 2 3
Vektor b: 1 2 3
Matrixzeile 0: 4 5 6
Matrixzeile 1: 7 8 9
a*b: 32  50
a.size()=2 b.size()=3
```

Konstanteniteratoren kommen beispielsweise zum Einsatz, wenn C++-Vektoren als konstante Referenzen übergeben werden.

Beispiel:

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

typedef

Mit `typedef` können weitere Namen für einen Datentyp vereinbart werden. Die Syntax entspricht der Variablenvereinbarung, wobei die Speicherklasse (später!) durch `typedef` und der Variablenname durch den Typnamen ersetzt wird. Zwar können mit `typedef` Namen für Funktionsdatentypen vereinbart werden, damit ist aber die Definition von Funktionen *nicht* möglich (im Unterschied zur Deklaration von Funktionen oder der Vereinbarung von Funktionszeigern).

Bsp.:

```
typedef unsigned int size_t                // Typnamenvereinbarung
size_t n;                                  // Variablenvereinbarung

typedef double myvector[10];               // Typnamenvereinbarung
myvector a;                                // Variablenvereinbarung

typedef double *doubleptr;                 // Typnamenvereinbarung
doubleptr xp;                              // Variablenvereinbarung

// Ab hier:  Funktionen und Funktionszeiger (spaeter!)
typedef void func(int), (*funcptr)(int);   // Typnamenvereinbarung:
void (*signal(int, void (*)(int)))(int);   // Funktionsdeklaration fuer signal
func *signal(int, func *);                 // aequivalente Funktionsdeklaration
funcptr signal(int, funcptr);              // aequivalente Funktionsdeklaration
func *signal(int, func *) { ... }          // Funktionsdefinition unzulaessig
funcptr signal(int, funcptr) { ... }       // Funktionsdefinition unzulaessig
```

Auch innerhalb von Klassen können neue Typnamen durch typedef vereinbart werden. Sie unterliegen den gleichen Zugriffsregeln wie Komponentennamen.

Bsp.: Klasse Vektor mit Iteratordefinitionen

```
class Vektor {
  private:
    double *ap;
    int    len;
  public:
    Vektor() : ap(0),len(0) { }                         // Vektor der Laenge 0
    Vektor(int n, double x=0): len(n) {                 // Vektor der Laenge n
      ap = new double [n];
      for (int i=0; i<n; ++i) ap[i]=x;
    }
                    :

    typedef double *iterator;
    typedef const double *const_iterator;
    typedef int size_type;

    iterator begin() { return ap;     }
    iterator end()   { return ap+len; }
    const_iterator begin() const { return ap;     }
    const_iterator end()   const { return ap+len; }

};
```

```
int main()
{
  int n;
  cout << "n: "; cin >> n;

  Vektor a(n);
  Vektor::iterator pos;

  // Eingabe von a:
  cout << "a[0] ... a[" << n-1 << "]: ";
  for (pos=a.begin(); pos!=a.end(); ++pos)
    cin >> *pos;

  // Ausgabe von a:
  for (pos=a.begin(); pos!=a.end(); ++pos)
    cout << *pos << "  ";
  cout << endl;

  const Vektor c(a);
  Vektor::const_iterator cpos;

  // Ausgabe von c:
  for (cpos=c.begin(); cpos!=c.end(); ++cpos)
    cout << *cpos << "  ";
  cout << endl;

  return 0;
}
```

Ausgabe:

```
n: 5
a[0] ... a[4]: 1 3 5 7 9
1  3  5  7  9
1  3  5  7  9
```