Bsp.: Polynome als doppelt verkettete Liste

```
#include <iostream>
#include <iomanip>
#include <string>
#include <list>
#include <utility>

using namespace std;
using namespace rel_ops;

struct Monom {
  int i;
  long a;

  Monom() :  i(0),a(0) {}

  int grad() { if (a!=0) return i; else return -1; }

  friend bool operator<(Monom m1, Monom m2) {   // Vergleich von Monomen nur nach Grad
    return m1.grad()>m2.grad();                // Sortierung nach absteigendem Grad
  }

  friend bool operator==(Monom m1, Monom m2) {  // Vergleich von Monomen nur nach Grad
    return m1.grad()==m2.grad();
  }
};


class Polynom
{
  private:
    list<Monom> ml;
  public:
    Polynom() {}

  friend Polynom operator+(Polynom p, Polynom q) {
    p.ml.merge(q.ml);
    list<Monom>::iterator pos, next;
    if (p.ml.empty()) return p;
    pos=next=p.ml.begin();
    ++next;                                   // moeglich, weil p.ml nicht leer
    for (; next!=p.ml.end(); ++pos, ++next) {
      if (pos->i==next->i) {
        pos->a+=next->a;
        next->a=0;
      }
    }
    Monom null;
    p.ml.remove(null);
    return p;
  }
```

```
  friend istream& operator>>(istream& stream, Polynom& p)
  // Eingabe darf keine Monome gleichen Grads beinhalten
  {
    char c1,c2;
    Monom m;
    p.ml.clear();
    while(stream >> m.a >> c1 >> c2 >> m.i) {
      if (c1!='x' || c2!='^') {
        // Fehlerbehandlung rudimentaer
        stream.setstate(ios::failbit);
        break;
      }
      p.ml.push_back(m);
      // Am Zeilenende zu lesen aufhoeren
      if (stream.peek()=='\n') {
        break;
      }
    }
    p.ml.sort();
    Monom null;
    p.ml.remove(null);
    return stream;
  }

  friend ostream& operator<<(ostream& stream, const Polynom& p)
  {
    if (p.ml.empty()) stream << "0";
    list<Monom>::const_iterator pos;
    for (pos=p.ml.begin(); pos!=p.ml.end(); ++pos) {
      stream << showpos   << pos->a << "x^";  // Koeffizient mit Vorzeichen ausgeben
      stream << noshowpos << pos->i;          // Exponent ohne Vorzeichen ausgeben
    }
    return stream;
  }
};


int main()
{
  Polynom p,q;
  cout << "p: "; cin >> p;
  if (!cin) { cerr << "Eingabefehler!" << endl; return 1; }
  cout << "q: "; cin >> q;
  if (!cin) { cerr << "Eingabefehler!" << endl; return 1; }
  cout << "p+q: " << p+q << endl;
  return 0;
}
```

Ausgabe:

```
p: 4x^3+2x^2+1x^1
q: 1x^3-2x^2+1x^0
p+q: +5x^3+1x^1+1x^0
```

Assoziative Vektoren (`<map>`)

Paarbildung (`<utility>`)

Im folgenden bezeichne (s,t) den aus dem Wert s vom Datentyp S und Wert t vom Datentyp T gebildeten Record vom Typ `struct{`S `first;` T `second;}`.

Operation	*Wirkung*
`pair<S,T> p`	vereinbart p mit den Voreinstellungen für S und T
`pair<S,T> p(s,t)`	vereinbart p und initialisiert es mit (s,t)
`make_pair(s,t)`	liefert (s,t)
`p.first`	liefert s
`p.second`	liefert t

Vereinbarung und Operationen assoziativer Vektoren

Die Reihenfolge der Komponenten von assoziativen Vektoren in C++ ergibt sich aus der Ordnung des Indextyps, für die der Operator `<` definiert sein muss oder durch das voreingestellte bzw. angegebene Funktionsobjekt aus einer Funktorklasse Cmp.

Die Elemente assoziativer C++-Vektoren werden als Paare abgespeichert, die jeweils aus einem Indexwert und einem Komponentenwert bestehen und den Datentyp `pair<const` I`,`T`>` haben.

Im folgenden steht I für den Indextyp, T für den Komponententyp, i für einen Wert vom Typ I, t für einen Wert vom Typ T, und p für einen Wert vom Typ `pair<const` I`,`T`>`, der z.B. durch `p=pair<const` I`,`T`>(`i`,`t`)` erzeugt werden kann.

Operation	*Wirkung*
`map<I,T> a  map<I,T,Cmp> a`	vereinbart leeren assoziativen Vektor, Ordnung
`map<I,T,Cmp> a(cmp)`	ggf. durch `cmp` induziert
`map<I,T> a(b)  map<I,T,Cmp> a(b)`	vereinbart assoziativen Vektor a als Kopie von b
`map<I,T> a={{i`$_0$`,t`$_0$`},...,{i`$_{n-1}$`,t`$_{n-1}$`}}`*	vereinbart assoz. Vektor a mit $a[i_k] = t_k$
`map<I,T> a{{i`$_0$`,t`$_0$`},...,{i`$_{n-1}$`,t`$_{n-1}$`}}`*	"
	analog für `map<I,T,Cmp> a ...`
`a[i]`	liefert Referenz auf die Komponente zum Index i; falls Index i nicht vorhanden, wird er zusammen mit einem nach Voreinst. initial. Komp.wert *erzeugt*
`a=b`	Zuweisung
`a={{i`$_0$`,t`$_0$`},...,{i`$_{n-1}$`,t`$_{n-1}$`}}`*	Zuweisung: a wird assoz. Vektor mit $a[i_k] = t_k$
`a==b a!=b a>b a<b a<=b a>=b`	Vergleiche
`a.size()`	Zahl der Komponenten
`a.max_size()`	maximale Zahl der Komponenten
`a.clear()`	löscht alle Indizes und Komponenten
`a.empty()`	wahr, falls assoziativer Vektor leer
`a.insert(p)`	fügt Index-Komponentenpaar p ein (Rückgabewert s.u.)
`a.erase(i)`	löscht Vektorkomponente mit Index i, liefert 1 im Erfolgsfall, sonst 0
`a.count(i)`	1, falls Komp. zum Index i vorhanden, 0 sonst
`a.swap(b)`	vertauscht Indizes und Komp. von a mit denen von b

*C++11/14

Iteratoren und Listenfunktionen

Der Datentyp `map` stellt vier *bidirektionale* Iteratoren zur Verfügung.

Für jede Durchlaufrichtung gibt es zwei Iteratoren, davon jeweils einen für Verweise auf konstante Komponenten.

Trotz des Vorhandenseins des Indexoperators `[]` sind diese Iteratoren *keine* random-access-Iteratoren.

Operation	*Wirkung*
`map<I,T>::iterator pos`	vereinbart Vorwärtsiteratorvariable *pos*
`map<I,T>::const_iterator pos`	vereinbart Konstantenvorwärtsiteratorvariable *pos*
`map<I,T>::reverse_iterator pos`	vereinbart Rückwärtsiteratorvariable *pos*
`map<I,T>::const_reverse_iterator pos`	vereinbart Konstantenrückwärtsiteratorvariable *pos*
`map<I,T> a(anf,end)`	vereinbart assoziativen Vektor und kopiert die Elemente,
`map<I,T> a(anf,end,Cmp)`	auf die der Iteratorbereich $[anf, end)$ verweist
`a.begin()`	liefert Vorwärtsiterator für das erste Index-Komp.paar
`a.end()`	liefert Vorwärtsiterator für die Position *nach* dem letzten Index-Komponentenpaar
`a.rbegin()`	liefert Rückwärtsiterator für das letzte Indexkomponentenpaar
`a.rend()`	liefert Rückwärtsiterator für die Position *vor* dem ersten Indexkomponentenpaar
`++pos pos++`	inkrementiert Iterator *pos*
`--pos pos--`	dekrementiert Iterator *pos*
`*pos`	liefert Referenz auf Paar (i, t)
`pos->first  pos->second`	liefert Referenz auf i bzw. t
`a.find(i)`	liefert Iterator *pos* zum Index i oder `a.end()`
`a.insert(p)`	fügt Index-Komponentenpaar p ein[†], liefert Paar $(pos, success)$ mit *success* vom Typ `bool`
`a.insert(pos0,p)`	fügt Index-Komponentenpaar p ein[†], *pos0* wird als Vorschlag für die Einfügeposition *pos* betrachtet; liefert *pos*
`a.insert(anf,end)`	fügt die Index-Komponentenpaare ein[†], auf die der Iteratorbereich [*anf*, *end*) verweist
`a.erase(pos)`	löscht Komponente und Index an der Position *pos*
`a.erase(anf,end)`	löscht Komponenten und Indizes, auf die der Iteratorbereich [*anf*, *end*) verweist

Bem.: `a[i]` liefert `(*(a.insert(make_pair(i,T())).first)).second`

[†] falls Index bzw. Indizes noch nicht in der map `a` vorhanden