Verkettete Listen

Bereits in C können Records Zeiger auf einen oder mehrere Records des eigenen Datentyps enthalten. Das ermöglicht den Aufbau einfach- oder doppelverketteter Listen und allgemeiner von baumartigen Datenstrukturen. In C++ gibt es in der STL eigene Datentypen für einige wichtige Datenstrukturen dieser Art.

Vereinbarung von Listenelementen

Eine Klassendefinition ist erst vollständig, wenn die rechte Mengenklammer } erreicht ist. Die Verwendung des Klassennamens ist daher innerhalb der Klassendefinition nur eingeschränkt möglich, beispielsweise ist die Vereinbarung einer Datenkomponente vom Klassentyp nicht erlaubt, ein Zeiger hierauf allerdings schon.

Bsp.: `class Element { int i; Element  e; }` *unzulässig*
`class Element { int i; Element *e; }` *zulässig*

Einfache oder doppelverkettete Listen kommen zum Einsatz, wenn der Zeitaufwand für das Einfügen oder Löschen einzelner Elemente nicht von der Gesamtanzahl der Elemente der Datenstruktur abhängen soll. Dünn besetzte Vektoren können gelegentlich als verkettete Liste effizienter verarbeitet werden.

Bsp.: Polynome als einfachverkettete Liste (ohne Verwendung eines Listendatentyps)

```
#include <iostream>
#include <iomanip>

using namespace std;

struct monom {int i; long a; monom *z;};

monom *negativ(monom *p)
{
  monom *r=p;
  while(p) {
    p->a = -p->a; p = p->z;
  }
  return r;
}

monom *lies(string s)
{
  int i; long a; char c1,c2;
  monom anf, *r=&anf , *ralt;
  anf.z = 0;
  cout << s << ": ";
  while(cin >> a >> c1 >> c2 >> i) {
    ralt = r;
    r = new monom;
    r->i = i; r->a = a; r->z = 0;
    ralt->z = r;
    if (cin.peek()=='\n') break;
  }
  return anf.z;
}
```

```
void schreibe(string s, monom *p)
{
  cout << s;
  if ( p==0 ) cout << "0";
  while (p) {
    cout << showpos  << p->a << "x^" << noshowpos << p->i;
    p = p->z;
  }
  cout << endl;
  return;
}


int main()
{
  monom *p;
  p = lies("p");
  schreibe("+p = ",p);
  negativ(p);
  schreibe("-p = ",p);
  return 0;
}
```

Ausgabe:

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

Doppelt verkettete Listen in der Standard Template Library (`<list>`)

Vereinbarung und Operationen

Im folgenden steht T für den Listenkomponententyp, t für ein Objekt vom Datentyp T, n ist eine vorzeichenlose Zahl (vom Typ `list<T>::size_type`).

Operation	*Wirkung*
`list<T> a`	vereinbart leere Liste
`list<T> a(n)`	vereinbart Liste aus n Komponenten, jeweils nach Voreinstellung für T initialisiert
`list<T> a(n,t)`	vereinbart Liste aus n Komponenten, jeweils mit t (vom Datentyp T) initialisiert
`list<T> a(b)`	vereinbart Liste a als Kopie von b
`list<T> a=`$\{$`t`$_0\ldots$,`t`$_{n-1}\}^*$	vereinbart Liste a mit den Komponenten $t_0, \ldots, t_{n-1}$
`list<T> a`$\{$`t`$_0\ldots$,`t`$_{n-1}\}^*$	“
`a.front() a.back()`	liefert erste bzw. letzte Komponente von a (falls vorhanden)
`a=b`	Zuweisung
`a=`$\{$`t`$_0\ldots$,`t`$_{n-1}\}^*$	Zuweisung: a wird Liste mit den Komponenten $t_0 \ldots, t_{n-1}$
`a.assign(n) a.assign(n,t)`	weist a n nach Voreinst. bzw. mit t initial. Komp. zu
`a==b a!=b a>b a<b a<=b a>=b`	Vergleiche
`a.size()`	Zahl der Komponenten
`a.resize(n) a.resize(n,t)`	ändert Komponentenzahl, ggf. Erzeugung und Initialisierung
`a.push_front(t) a.push_back(t)`	Komponente mit Wert t am Anfang bzw. am Ende anhängen
`a.pop_front() a.pop_back()`	erste bzw. letzte Komponente löschen
`a.clear()`	alle Komponenten löschen
`a.empty()`	wahr, falls Liste leer
`a.remove(t)`	löscht alle Komponenten mit Wert t
`a.remove_if(pred)`	löscht alle Komponenten, für die *pred*(t) wahr ist
`a.swap(b)`	vertauscht Komponenten von a mit denen von b
`a.sort()`	sortiert Liste gemäß `<`
`a.sort(cmp)`	sortiert Liste gemäß *cmp*: *cmp*(t_1, t_2) wahr induziert Ordnung $<$, so daß entweder $t_1 < t_2$, $t_2 < t_1$ oder $t_1 \sim t_2$ gilt, wobei $\sim$ eine Äquivalenzrelation ist.
`a.merge(b)`	sortiert b in a ein gemäß $<$ (a, b zuvor sortiert, b danach leer)
`a.merge(b,cmp)`	wie `a.merge(b)`, jedoch Sortierung gemäß *cmp*
`a.reverse()`	Reihenfolge umkehren
`a.unique()`	entfernt alle direkt aufeinanderfolgenden Komponenten mit demselben Wert bis auf das erste
`a.unique(equal)`	wie `a.unique()`, mit t_1 gleich t_2, falls *equal*(t_1, t_2) wahr.

Im Unterschied zum Datentyp `vector` fehlt der Komponentenzugriff mittels `a[i]`, andererseits ist das Verlängern und Verkürzen der Liste auch am Anfang möglich.

Bsp.: Eingabezeilen sortiert ausgeben

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

using namespace std;

int main()
```

*C++11/14

```
{
  string zeile;
  list<string> puffer;

  while (getline(cin,zeile))
    puffer.push_back(zeile);

  puffer.sort();

  while(!puffer.empty()) {
    cout << puffer.front() << endl;
    puffer.pop_front();
  }

  return 0;
}
```

Iteratorfunktionen

Der `list`-Datentyp stellt vier bidirektionale Iteratoren bereit. Ein wahlfreier Zugriff ist mit ihnen *nicht* möglich.

Operation	*Wirkung*
`list<T>::iterator pos`	Vorwärtsiteratorvariable *pos* vereinbaren
`list<T>::const_iterator pos`	Konstantenvorwärtsiteratorvariable *pos* vereinbaren
`list<T>::reverse_iterator pos`	Rückwärtsiteratorvariable *pos* vereinbaren
`list<T>::const_reverse_iterator pos`	Konstantenrückwärtsiteratorvariable *pos* vereinbaren
`a.begin()`	liefert Vorwärtsiterator für die erste Komponente
`a.end()`	liefert Vorwärtsiterator für die Position *nach* der letzten Komp.
`a.rbegin()`	liefert Rückwärtsiterator für letzte Komponente
`a.rend()`	liefert Rückwärtsiterator für die Position *vor* der ersten Komp.
`++pos pos++ --pos pos--`	inkrementiert bzw. dekrementiert Iterator *pos*
`*pos`	Komponentenwert zur Iteratorposition *pos*
`list<T> a(anf,end)`	vereinbart Liste, initialisiert sie Werten, auf die der der Iteratorbereich [*anf*, *end*) verweist
`a.assign(anf,end)`	weist der Liste *a* als Komponenten die Werte zu, auf die der der Iteratorbereich [*anf*, *end*) verweist
`a.insert(pos)`	fügt an der Position *pos* eine neue nach Voreinstellung initialisierte Komponente ein; liefert Position der neuen Komp.
`a.insert(pos,t)`	fügt an der Position *pos* eine neue mit dem Wert *t* initialisierte Komponente ein; liefert Position der neuen Komp.
`a.insert(pos,anf,end)`	fügt an der Position *pos* Kopien der Komp. ein, auf die der Iteratorbereich [*anf*, *end*) verweist
`a.erase(pos)`	löscht Komponente an der Position *pos*; liefert Position der folgenden Komponente
`a.erase(anf,end)`	löscht Komponenten im Bereich [*anf*, *end*), liefert Folgepos.
`a.splice(pos,b)`	fügt Liste *b* an der Position *pos* in *a* ein, *b* danach leer
`a.splice(pos,b,anf)`	fügt Komp. zu *anf* von Liste *b* an Position *pos* in *a* ein, *b* ist entsprechend verkürzt
`a.splice(pos,b,anf,end)`	fügt Komponenten zu [*anf*, *end*) an Position *pos* in *a* ein, *b* ist entsprechend verkürzt