§2 Überladen II – STL-Behälterklassen

Leitideen: *Die STL-Behälterklassen stellen eine Reihe gleichnamiger Funktionen und Operatoren für unterschiedliche Behältertypen bereit. In der Regel werden für einen Behältertyp nur effizient realisierbare Operationen angeboten.*

Zum Durchwandern der Behälter gibt es Iteratoren (verallgemeinerte Zeiger), die sich je nach Durchlaufrichtung und Konstanz der Zielobjekte unterscheiden. Den Iteratorbereichen liegt die Idee halboffener Intervalle zugrunde.

Bei Zuweisungen und Initialisierungen wird Wertsemantik verwendet, d.h. die Behälterkomponenten werden kopiert.

Bei konstanten Behältern sind die Behälterkomponenten konstant.

§2 Überladen II – Themenübersicht

- Iteratoren - Konzept I,II,III
- Vereinbarung neuer Typnamen mit typedef
- Verkettete Listen I,II,II
- Listen in der STL I,II
- Polynome als Liste
- Assoziative Vektoren I,II
- Assoziative Vektoren - Beispiele I,II
- Fehlerbehandlung mit Exceptions *(Zusatzfolie!)*
- Funktionszeiger und Behälter *(Zusatzfolie!)*

Iteratoren - Konzept

- *Iteratoren* sind verallgemeinerte Zeiger zum Durchlaufen der STL-Behälter
- Spezifikation von Durchlaufsbereichen als halboffene Intervalle

```
double a[N], s=0; int i;
for (i=0; i<N; ++i) s += a[i];
// halboffenes Intervall [0,N[ in ℤ

double *p;
for (p=a; p!=a+N; ++p) s += *p;
// halboffenes Intervall [a,a+N[ in ℤ

vector<double> a;
vector<double>::iterator pos;
for (pos=a.begin(); pos!=a.end(); ++pos)
   s += *pos
// halboffenes Intervall [a.begin(),a.end()[
```

Durchlaufrichtung $\longrightarrow$

Iteratoren - Konzept II

Operatoren

`++pos pos++ --pos pos--`	für bidirektionale Iteratoren
`*pos pos->comp`	“
`pos+i i+pos pos-i`	zusätzlich für Iteratoren
`pos+=i pos-=i`	mit wahlfreiem Zugriff
`pos[i]` [= `*(pos+i)`]	(Random-Access-Iteratoren)

Rückwärtsiteratoren (Umgekehrte Durchlaufrichtung)

- ```
  vector<double>::reverse_iterator rpos;
  for (rpos=a.rbegin(); rpos!=a.rend(); rpos++)
      s += *rpos
  ```
- Halboffenes Intervall fur Rückwärtsiteratoren:
  `]a.rend(),a.rbegin()]`
  Durchlaufrichtung ⟵
- *Unterschiedliche* Angaben für Intervallenden bei Vorwärts- und Rückwärtsiteratoren, obwohl dieselben Elemente durchlaufen werden
  ```

Iteratoren - Konzept III

Konstanteniteratoren

- Konstante Iteratoren als Nachbildung von `const T *pc` (Zeiger auf Konstante)
- *Unzulässig:* `T *p; const T *pc; p=pc;`
- Komponentenfunktionen mit `const`-Attribut erlauben unterschiedliche Behandlung von konstanten und nicht konstanten Objekten
- Dadurch Realisierung des Zugriffsschutzes für konstante STL-Behälter:
  Ähnlich wie bei einem konstanten C-Vektor sollen auch bei einem konstanten STL-Vektor die Komponenten konstant sein
- Unterschied wichtig bei Übergabe von Referenzen auf Konstanten
- Insgesamt also 4 Iteratorentypen in STL-Behältern:
  Vorwärts- und Rückwärtsiteratoren,
  konstante Vorwärts- und konstante Rückwärtsiteratoren

Vereinbarung neuer Typnamen mit typedef

- Variablenvereinbarungen kann eine weitere Deklarationsangabe (Speicherklasse) vorangestellt sein *(später!)*
- Syntax der Typnamenvereinbarung wie bei der Variablenvereinbarung:
  Speicherklasse → `typedef`
  Variablenname → Typname
- Auch innerhalb von Klassen möglich, Zugriff außerhalb über *C*`::`*Typname* (→ *Inf.bl.7)*

Beispiele:

`extern unsigned int s`	s: `unsigned int`
`typedef unsigned int size_t`	size_t: `unsigned int`
`extern double a[10]`	a: 10-Vektor von double
`typedef double myvector[10]`	myvector: 10-Vektor von double
`extern double *xp`	xp: Zeiger auf `double`
`typedef double *doubleptr`	doubleptr: Doublezeigertyp

Verkettete Listen

- Record (Klasse) kann Zeiger auf eigenen Datentyp enthalten, dadurch einfach- oder doppelverkettete Listen bzw. Bäume möglich
- syntaktisch: unvollständige Typvereinbarung – Zeiger auf eigenen Typ möglich, nicht aber Datenkomp. dieses Typs

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

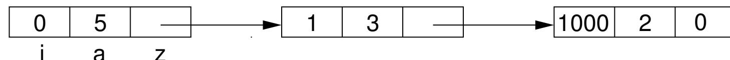
- Vorteil: schnelles Einfügen und Löschen, effizientere Implementierung dünnbesetzter Vektoren und Matrizen

Beispiel:

Polynom: $5x^0 + 3x^1 + 2x^{1000}$

Datentyp eines Summanden:

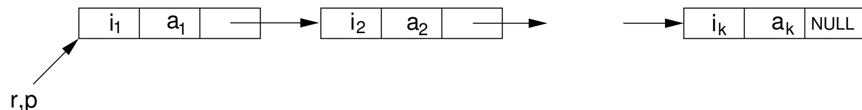
```
struct monom {int i; long a; monom *z;};
```



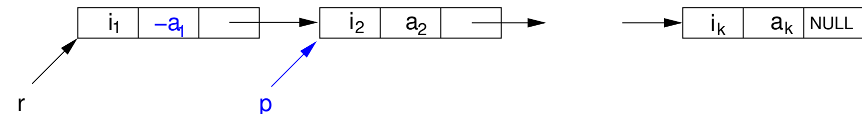
Verkettete Listen II

Funktion negativ:

vor dem 1. Schritt:



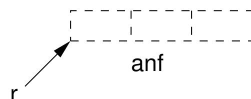
nach dem 1. Schritt:



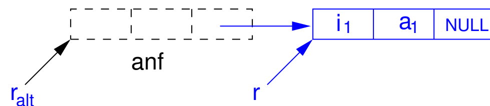
Verkettete Listen III

Funktion lies:

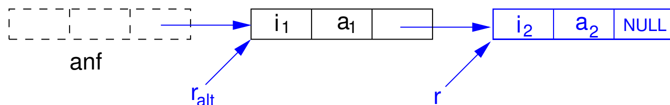
vor dem 1. Schritt:



nach dem 1. Schritt:



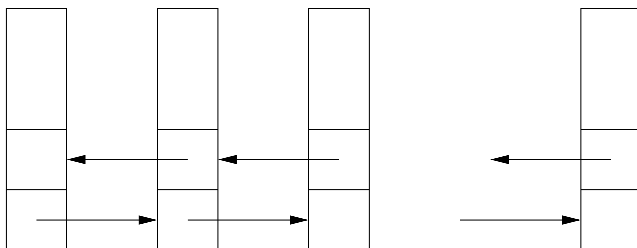
nach dem 2. Schritt:



Listen in der Standard Template Library (STL)

Eigenschaften

- Vereinbarung: `list<T> a`
  a leere Liste mit Komponententyp *T*
- Einfügen/Löschen sind Operationen mit konstantem Zeitaufwand (falls Position gegeben)
- Implementierung als doppelt verkettete Liste naheliegend



- Iteratoren bidirektional, *kein* wahlfreier Zugriff, insbesondere *kein* Komponentenzugriff mittels `a[i]`

Listen in der STL - Fortsetzung

Weitere Eigenschaften

- Verlängern und Verkürzen sowohl am Anfang als auch am Ende möglich:
  `a.push_front(t),a.pop_front(),`
  `a.push_back(t), a.pop_back()`
- Häufig gebrauchte Listenoperationen als Komp.funktionen vorhanden:
  Einfügen, Löschen, Sortieren, Verschmelzen (merge), Spleißen (splice)
- Effizient implementierbare Operationen wie Vertauschen und Umkehren zusätzlich als Komponentenfunktionen vorhanden:
  `a.swap(b),a.reverse()`

Polynomaddition durch Listenoperationen - Beispiel

Polynome nach Grad vorsortiert:

$p: \quad 4x^3 \quad +2x^2 \quad +x$

$q: \quad x^3 \quad -2x^2 \quad +1$

Verschmelzen beider Listen:

$p: \quad 4x^3 \quad +x^3 \quad +2x^2 \quad -2x^2 \quad +x \quad +1$

$q:$ leer

Addition der Monome gleichen Grads (zweites Monom danach 0):

$p: \quad 5x^3 \quad +0x^3 \quad +0x^2 \quad +0x^2 \quad +x \quad +1$

$q:$ leer

Entfernen der Nullmonome:

$p: \quad 5x^3 \quad +x \quad +1$

$q:$ leer

Polynome mittels C++-Listen - Anmerkungen

- Monom ax^i als Klasse mit Komponenten i und a speichern
- Grad von Nullmonomen $0x^i$: -1
- Vergleichsoperatoren $<$ und $==$ zwischen Monomen:
  Nur Grad betrachten
- Polynom: Klasse basierend auf Liste von Monomen,
  absteigend nach Grad sortiert
- Nullpolynom: leere Liste   Nullmonome: $0x^i$
- Polynomaddition: Verschmelzen beider sortierter Listen, jeweils zwei Monome gleichen Grades addieren (d.h. hier Koeffizientensumme in das eine schreiben und das andere zum Nullmonom machen), Nullmonome entfernen
- Bereits beim Einlesen Monome sortieren und Monome mit Wert 0 entfernen
  Voraussetzungen: Eingabe enthält nur Monome von unterschiedlichem Grad, ein Polynom pro Zeile

Assoziative Vektoren

Allgemeines

Vektor mit Indexbereich $\neq \{0, 1, \ldots, n-1\}$ (allg. $\neq [a, b] \cap \mathbb{Z}$)

Bsp.:

```
guthaben["Maier"]    =   1;
guthaben["Maurer"]    =  5;
guthaben["Meyer"]    =  12;
guthaben["Moser"]    =   3;
guthaben["Mueller"] =   3;
```

Allg.: $a[i_0] = t_0$ $\quad$ $i_0, \ldots i_{n-1} \in I$ (Indexdatentyp)

$\vdots$ $\quad$ $i_0, \ldots i_{n-1}$ paarweise verschieden

$a[i_{n-1}] = t_{n-1}$ $\quad$ $t_0, \ldots t_{n-1} \in T$ (Komp.datentyp)

Math.: Funktion $a : \{i_0, \ldots, i_{n-1}\} \to T$, $a[i_k] = t_k$
Darstellung dieser Funktion durch ihren Graph
$\{(i_0, t_0), \ldots, (i_{n-1}, t_{n-1})\}$

Implementierungsaspekte

1. Schnelle Feststellung, ob $i \in \{i_0, \ldots, i_{n-1}\}$
2. Falls i_k gefunden, sollte t_k rasch bestimmbar sein
3. Rasches Einfügen und Löschen

Assoziative Vektoren - Fortsetzung

Idee: Falls $i_0, \ldots, i_{n-1}$ sortiert, Binärsuche möglich
Aufwand für die Suche: $O(\ln n)$ statt $O(n)$

Datenstruktur: balancierter Binärbaum (Rot-Schwarz-Baum)

Eigenschaften des Datentyps map

- Vereinbarung: map<I,T> a leere Map
  kann Elemente vom Typ pair<const I,T> aufnehmen
- Indextyp I: muss sortierbar sein („strikt schwache“ Ordnung bzgl. $<$, d.h. es gilt immer genau eine der Beziehungen $i < i'$, $i' < i$, $i \sim i'$ mit $\sim$ Äquivalenzrelation)
- Komponentenzugriff $a[i]$: $O(\ln n)$ Zugriffszeit
  Index i nicht vorhanden $\Longrightarrow$ Einfügen von (i, t_0) in Map
  (t_0: voreingestellter Wert für Typ T, in der Regel 0 o.ä)
- Deshalb: Bei Suche a.find(i) statt a[i] benutzen
- Nur bidirektionale Iteratoren, *nicht* random access.

Assoziative Vektoren - Beispiele

Resourcenverbrauch

- Map a (Teilnehmer `string`, Verbrauch `int`) anlegen - zunächst leer
- `a[s]+=n` Einfügen von $(s, 0)$ in `a`, sofern s noch kein Schlüssel, Addition von n bewirkt Änderung zu (s, n)
- Durchwandern von `a` mit Iterator `pos`:
  Schlüssel `pos->first` , Wert `pos->second`

Erfassung von Übungspunkten

- Teilnehmer aus Datei lesen, durch `a[s]` als $(s, 0)$ in Map a einfügen
- Schutz vor Tippfehlern bei interaktiver Eingabe:
  `a.find(s)` statt `a[s]`
- Neuen Teilnehmer und Punktzahl zu Demonstrationszwecken mit `insert` angelegen

Assoziative Vektoren - Beispiele II

Worthäufigkeit

- Map `haeufigkeit` (Wort `string`, Häufigkeit `int`) anlegen – zunächst leer
- Wort zählen mit `haeufigkeit[wort]++` – neu angelegtes Wort hat Häufigkeit 1
- Zweiter Behälter `vector` mit Werten (*wort*, *n*) als Kopie der Map `haeufigkeit`
- Umsortieren mit `sort` unter Verwendung der booleschen Funktion `ordnung` – induziert strikt schwache Ordnung auf `pair<string,int>` (absteigende Häufigkeit)
- Besser als `sort` ist `stable_sort` wegen Erhaltung der alphabetischen Sortierung
- Durchwandern des Behälters `vector` statt mit Iterator auch mit Index möglich