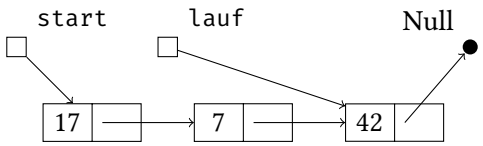
Verzeigerte Datenstrukturen

Bereits in C können records Zeiger auf Werte des eigenen Datentyps enthalten. Das ermöglicht den Aufbau von graph-, baum- bzw. listenartigen Datenstrukturen wobei die Zeiger als gerichtete Kanten fungieren. In C++ gibt es in der STL Datentypen für einige der wichtigeren Datenstrukturen dieser Art.

Wichtige Einschränkung ist, dass sowohl structs in C wie auch Klassen in C++ keine Werte ihres eigenen Datentyps direkt als Felder bzw. Attribute enthalten können. Hinreichendes Argument ist, dass Datentypen sowohl in C wie auch in C++ eine wohldefinierte und endliche Größe im Sinne des von ihnen benötigten Speichers haben müssen. Die Formulierung der relevanten Fehlermeldung stützt sich darauf, dass die Definition eines structs bzw. einer Klasse erst vollständig ist, sobald die schließende rechte geschweifte Klammer } erreicht ist. Sehr wohl erlaubt innerhalb von structs bzw. Klassen ist jedoch das Vorkommen von Zeigern auf den eigenen Datentyp, insb. da diese ja eine wohldefinierte Größe haben (oft genau 8 byte).

Beispiel (Einfach verkettete Listen). Wir vereinbaren eine eigene Klasse zur Speicherung einer beliebig langen Liste von `double` Werten. Hierbei identifizieren wir jedes Element der Liste mit einem Objekt der Klasse `Liste` wobei jedes Objekt sowohl einen Wert vom Typ `double` als Attribut trägt, wie auch einen Zeiger auf das direkt folgende nächste Element der Liste.



Speicherlayout einer einfach verketteten Liste

Das Beispiel ist unvollständig. Es fehlt insbesondere der Destruktor; so wie dargestellt verursacht Verwendung der Klasse ein Speicherleck.

list.cpp

```cpp
#include <iostream>

using namespace std;

class Liste {
  public:
    double v;

  private:
    Liste* next;

  public:
    explicit Liste(double v_ = 0): v(v_), next(nullptr) {}

    static Liste* einlesen(string n) {
      Liste *start = nullptr, *lauf = start;

      cout << n << ": ";

      double x;
      while (cin >> x) {
        Liste* elem = new Liste{x};
```

```
        if (!start)
          start = lauf = elem;
        else {
          lauf->next = elem;
          lauf = elem;
        }
        if (cin.peek() == '\n') break;
      }

      return start;
    };

    friend ostream& operator<<(ostream& stream, const Liste& l) {
      for (const Liste *lauf = &l; lauf; lauf = lauf->next)
        stream << lauf->v << " ";
      return stream;
    }
};

int main() {
  Liste *liste = Liste::einlesen("l");
  if (!liste)
    return 2;

  cout << "l: " << *liste << endl;
  return 0;
}
```

```
l: 17 7 42 13
l: 17 7 42 13
```

STL-Listen (`list`)

Objekte vom Datentyp `list<T>` speichern beliebig viele Elemente vom Typ T als *doppelt* verkettete Liste. D.h. jedes Element speichert sowohl einen Zeiger auf das nächste, wie auch das vorherige Elemente in der Liste. Das ermöglicht es manche Operationen effizienter zu implementieren und vor Allem bidirektionale Iteratoren anbieten zu können.

Einfach verkettete Listen existieren in der STL ebenfalls unter dem Namen `forward_list`.

Im Gegensatz zu `vector` erlaubt `list` nicht den wahlfreien Zugriff auf Elemente gegeben einen numerischen Index. Insb. implementieren deswegen Iteratoren über Werte vom Typ `list` einige Operationen nicht (z.B. i `-` i' für Iteratoren i und i').

Im Folgenden bezeichnet T einen Datentyp, $t, t_0, \ldots, t_{n-1}$ Werte vom Typ T, n eine natürliche Zahl vom Typ `list<T>::size_type`, l, l' Variablen vom Typ `list<T>`, p der Name einer Funktion vom Typ `bool` p`(T t)` und c, e der Name von Funktionen vom Typ `bool` c`(const T& t0, const T& t1)`. Zudem bezeichnen i, i' Iteratoren über l und j, j' Iteratoren über l'.

`list<T> l` Vereinbart Variable l vom Typ `list<T>` leer (Länge 0)

`list<`T`> `l`(`n`)`	Vereinbart Variable l vom Typ `list<`T`>` mit Länge n und allen Elementen initialisiert mit Standardkonstruktor
`list<`T`> `l`(`n`, `t`)`	Vereinbart Variable l vom Typ `list<`T`>` mit Länge n und allen Elementen initialisiert als Kopien von t
`list<`T`> `l`(`l'`)`	Vereinbart Variable l vom Typ `list<`T`>` als Kopie von l'
`list<`T`> `l`{`t_0`, ..., `t_{n-1}`}`	Vereinbart Variable l vom Typ `list<`T`>` mit Elementen $t_0, \ldots, t_{n-1}$
l`.front()` l`.back()`	Liefert erstes bzw. letztes Element von l falls vorhanden, sonst undefiniertes Verhalten
l `==` l' l `!=` l' l `>` l' l `<` l' l `<=` l' l `>=` l'	Vergleicht l und l' lexikographisch
l`.size()`	Liefert Anzahl der Elemente in l
l`.resize(`n`)` l`.resize(`n`, `t`)`	Ändert Anzahl der Elemente in l, ggf. Initialisierung neuer Elemente mit Standardkonstruktor bzw. als Kopie von t
l`.push_front(`t`)` l`.push_back(`t`)`	Fügt Kopie von t am Anfang bzw. Ende der Liste l ein
l`.pop_front()` l`.pop_back()`	Löscht erstes bzw. letztes Element von l
l`.clear()`	Löscht alle Elemente aus l
l`.empty()`	Liefert `true` g.d.w. l leer ist
l`.remove(`t`)` l`.remove_if(`p`)`	Löscht alle Elemente aus l die unter `operator==` äquivalent sind zu t bzw. für die p `true` liefert
l`.sort()` l`.sort(`c`)`	Sortiert die Elemente in l aufsteigend nach `operator<` bzw. c. Die Sortierung ist *stabil*, äquivalente Elemente behalten ihre Reihenfolge relativ zueinander.
l`.merge(`l'`)` l`.merge(`l'`, `c`)`	Sortiert, sofern l und l' bereits sortiert, l' unter Erhalt der Sortierung bzgl. `operator<` bzw. c in l ein. l' ist danach leer.
l`.reverse()`	Kehrt Reihenfolge der Elemente in l um
l`.unique()` l`.unique(`e`)`	Entfernt aus l alle direkt aufeinanderfolgenden Elemente die äquivalent sind bzgl. `operator==` bzw. e außer dem jeweils Ersten
`list<`T`>::iterator `i `list<`T`>::const_iterator `i `list<`T`>::reverse_iterator `i `list<`T`>::const_reverse_iterator `i	Vereinbart Variable i als Vorwärts-, Konstanten-, Rückwärts- bzw. Rückwärts-Konstanten-Iterator
l`.begin()` l`.end()` l`.rbegin()` l`.rend()`	Liefert Vorwärts-Iterator über l zum ersten bzw. *nach* dem letzten Element bzw. Rückwarts-Iterator über l zum letzten bzw. *vor* dem ersten Element
`++`i i`++` `--`i i`--`	Verschiebt Iterator i auf nächste bzw. vorherige Position in Durchlaufrichtung
`*`i	Referenz zu Wert an aktueller Position von i
`list<`T`> `l'`(`i`, `i'`)`	Vereinbart Variable l' vom Typ `list<`T`>` als Kopie der Elemente aus l im Bereich $[i, i')$
l`.insert(`i`)` l`.insert(`i`, `t`)`	Fügt bevor der aktuellen Position von i eine Komponente in l ein initialisiert mit Standardkonstruktor bzw. als Kopie von t

`l'.insert(j, i, i')`	Fügt bevor der aktuellen Position von i in l' Kopien ein der Elemente aus l im Bereich $[i, i')$
`l.erase(i)` `l.erase(i, i')`	Löscht aus l die Elemente an der aktuellen Position von i bzw. im Bereich $[i, i')$. Liefert Iterator auf Element das direkt auf letztes nun gelöschtes Element gefolgt hat.

`l.splice(i, l')`
`l.splice(i, l', j)`
`l.splice(i, l', j, j')`
	Verschiebt Elemente aus l' bzw. aus l' ab aktueller Position von j bzw. aus l' aus dem Bereich $[j, j')$ vor aktuelle Position von i in l. Derart verschobene Elemente sind hinterher nicht mehr vorhanden in l'.

Beispiel. Wir implementieren ein Hautprogramm das zunächst beliebig viele Zeilen von der Standardeingabe einliest und sie dabei in einer Liste vom Typ `list<string>` speichert. Danach wird die Liste sortiert und die Elemente der Liste, in nun lexikographisch sortierter Reihenfolge, wieder ausgegeben.

sortlist.cpp

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

using namespace std;

int main() {
  list<string> buffer;

  for (string line; getline(cin, line); )
    buffer.push_back(line);

  buffer.sort();

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

  return 0;
}
```

```
eins
zwei
drei
drei
eins
zwei
```

Beispiel. Wir implementieren eine Klasse `Polynom` zur Modellierung von Polynomen über $\mathbb{R}$ (approximiert durch `double`) intern als verkettete Liste von Monomen, d.h. Ausdrücken der Form $v \cdot x^p$.

listpol.cpp

```cpp
#include <iostream>
#include <list>
#include <utility>

using namespace std;
using namespace rel_ops;

class Monom {
  public:
    int pot;
    double v;

    Monom(double v_ = 0, int pot_=1): pot(pot_), v(v_) {}

    int grad() const { if (v != 0) return pot; else return -1; }

    friend bool operator<(const Monom& m1, const Monom& m2) {
      return m1.grad() > m2.grad();
    }
    friend bool operator==(const Monom& m1, const Monom& m2) {
      return m1.grad() == m2.grad();
    }

    friend ostream& operator<<(ostream& stream, const Monom& m) {
      return stream << showpos << m.v << "x^"
                    << noshowpos << m.pot;
    }
};

class Polynom {
  private:
    list<Monom> ml;

  public:
    Polynom() {}
    Polynom(double v_, int pot_=1): ml({Monom{v_, pot_}}) {}

    Polynom& operator+=(Polynom q) {
      ml.merge(q.ml);
      if (ml.empty()) return *this;
      list<Monom>::iterator curr = ml.begin(),
        next = ++(ml.begin());
      for (; next != ml.end(); ++curr, ++next) {
        if (curr->pot == next->pot) {
          curr->v += next->v;
          *next = Monom{};
        }
      }
      ml.remove(Monom{});
      return *this;
    }

    friend ostream& operator<<(ostream& stream, const Polynom& p) {
      if (p.ml.empty()) return stream << "0";
      for (const auto& monom: p.ml)
```

```
        stream << monom;
      return stream;
    }
};

int main() {
  Polynom p, q;
  p += Polynom{4, 3}; p += Polynom{2, 2}; p += Polynom{1, 1};
  q += Polynom{1, 3}; q += Polynom{-2, 2}; q += Polynom{1, 0};

  cout << "p+q: " << (p+=q) << endl;

  return 0;
}
```

```
p+q: +5x^3+1x^1+1x^0
```

STL-Paare (`pair` aus `<utility>`)

Im Folgenden bezeichnen T_1, T_2 Datentypen, t_1, t_2 Werte vom Typ T_1 bzw. T_2 und p eine Variable vom Typ `pair<`T_1`, `T_2`>`.

`pair<`T_1`, `T_2`> `p `pair<`T_1`, `T_2`> `p{t_1`, `t_2`}`	Vereinbart Variable p vom Typ `pair<`T_1`, `T_2`>` mit beiden Komponenten initialisert mit jeweiligem Standardkonstruktor bzw. als Kopie von t_1 bzw. t_2
`make_pair(`t_1`, `t_2`)`	Liefert (t_1, t_2) als Wert vom Typ `pair<`T_1`, `T_2`>`
p`.first` ` ` p`.second`	Zugriff auf Komponenten von p

Generische STL-Mengen (`set`)

Objekte vom Datentyp `set<`T`>` speichern beliebig viele Elemente vom Typ T *ohne Duplikate* in einer geeigneten Baumstruktur für effizienten Zugriff. Mengenelemente lassen sich nicht verändern, sondern nur einfügen und löschen. Um die interne Baumstruktur bilden zu können muss `operator<` geeignet überladen sein für T.

Im Folgenden bezeichnet T einen Datentyp, t, $t_0, \ldots, t_{n-1}$ Werte vom Typ T und s, s' Variablen vom Typ `set<`T`>`. Zudem bezeichnen i, i' Iteratoren über s.

`set<`T`> `s`  set<`T`> `s{`$t_0$`, ..., `$t_{n-1}$`}`	Vereinbart Variable s vom Typ `set<`T`>` leer bzw. mit Elementen $t_0, \ldots, t_{n-1}$
s` == `s'`  `s` != `s'`  `s` > `s'`  `s` < `s'` $s$` <= `$s'$` `$s$` >= `$s'$`	Vergleicht s und s' lexikographisch
s`.size()`	Liefert Anzahl der Elemente in s

`s.clear()`	Löscht alle Elemente aus s
`s.empty()`	Liefert `true` g.d.w. s leer ist
`s.count(`t`)`	Liefert `1` falls s Element enthält, dass äquivalent ist zu t, sonst `0`
`s.insert(`t`)`	Fügt Kopie von t ein in s. Liefert Paar (`pair`) aus Iterator auf Position von t in s (bzw. Element äquivalent zu t) und Wert vom Typ `bool` der angibt ob Einfügen stattgefunden hat (also kein Element äquivalent zu t bereits vorhanden war in s)
`s.erase(`t`)`	Löscht Element äquivalent zu t aus s, falls vorhanden. Liefert Anzahl gelöschter Elemente (`0` oder `1`)
`set<`T`>::iterator` i `set<`T`>::reverse_iterator` i	Vereinbart Variable i als Vorwärts-, bzw. Rückwärts-Iterator
`s.begin()` `s.end()` `s.rbegin()` `s.rend()`	Liefert Vorwärts-Iterator über s zum ersten bzw. *nach* dem letzten Element bzw. Rückwarts-Iterator über s zum letzten bzw. *vor* dem ersten Element
`s.find(`t`)`	Liefert Iterator zur Position von Element äquivalent zu t in s falls vorhanden, sonst `s.end()`
`s.lower_bound(`t`)` `s.upper_bound(`t`)`	Liefert Iterator zu erstem Element das *nicht kleiner* bzw. größer ist als t
`++`i i`++` `--`i i`--`	Verschiebt Iterator i auf nächste bzw. vorherige Position in Durchlaufrichtung
`*`i	Konstante Referenz zu Wert an aktueller Position von i
s'`.insert(`i`,` i'`)`	Fügt Kopien der Elemente aus s im Bereich $[i, i')$ ein in s'
`s.erase(`i`)` `s.erase(`i`,` i'`)`	Löscht Element an aktueller Position von i bzw. im Bereich $[i, i')$ aus s. Liefert Iterator auf Element das direkt auf letztes nun gelöschtes Element gefolgt hat.

Beispiel (Sieb des Eratosthenes). Wir implementieren ein Hauptprogramm, dass das Sieb des Eratosthenes verwendet um Primzahlen kleiner als 50 zu finden und auszugeben. Wir verwenden hierbei ein Objekt vom Typ `set<unsigned int>` um Zahlen zu speichern für die wir bereits ausschließen konnten, dass diese prim sind.

primes_set.cpp

```cpp
#include <set>
#include <iostream>

using namespace std;

const unsigned int n = 50;

int main() {
  set<unsigned int> nonprimes{0, 1};

  for (unsigned int k = 2; k*k < n; k++)
    if (!nonprimes.count(k))
      for (unsigned int i = k; i*k < n; i++)
```

```
        nonprimes.insert(i*k);

  for (unsigned int k = 0; k < n; k++)
    if (!nonprimes.count(k))
      cout << k << " ";
  cout << endl;

  return 0;
}
```

```
2 3 5 7 11 13 17 19 23 29 31 37 41 43 47
```

STL-Mengen natürlicher Zahlen (bitset)

Ein Objekt vom Datentyp `bitset<N>` speichert, für beliebies aber festes $N \in \mathbb{N}$, eine Folge von N Bits. Es wird also jeder Zahl $n \in [0, N) \subset \mathbb{N}$ jeweils ein Wert vom Typ `bool` zugeordnet.

Für Zwecke der Umwandlung zwischen `bitset<N>` und `string` und für Zwecke der arithmetischen shift-Operatoren wird das `bitset` verstanden als Folge von bits mit den niedrigsten Indizes *rechts*.

Beispiel. Wir demonstrieren die Umwandlung zwischen `bitset<8>` und `string`.

bitset_order.cpp

```
#include <iostream>
#include <bitset>
#include <cstddef>

using namespace std;

int main() {
  bitset<8> s("11110000");

  for (size_t i = 0; i < 8; i++)
    cout << "s[" << i << "]: " << s[i] << endl;

  return 0;
}
```

```
s[0]: 0
s[1]: 0
s[2]: 0
s[3]: 0
s[4]: 1
s[5]: 1
s[6]: 1
s[7]: 1
```

Im Folgenden bezeichne N eine „hinreichend konstante“ natürliche Zahl, s, s' Variablen vom Datentyp

`bitset<N>`, u eine vorzeichenlose ganze Zahl vom Typ `unsigned long`, t eine C++-Zeichenkette (`string`) bestehend nur aus den Zeichen `0` und `1`, j, k vorzeichenlose ganze Zahlen vom Typ `size_t` (aus `<cstddef>`), b eine Variable vom Typ `bool`, o ein Ausgabe- und i ein Eingabestrom.

`bitset<N> s`	Vereinbart Variable s vom Typ `bitset<N>` mit allen bits `0`
`bitset<N> s{u}`	Vereinbart Variable s vom Typ `bitset<N>` initialisiert anhand von u

`bitset<N> s{t}`
`bitset<N> s{t, j}`
`bitset<N> s{t, j, k}`
	Vereinbart Variable s vom Typ `bitset<N>` initialisiert anhand von t bzw. den Zeichen aus t ab Index j bzw. k vielen Zeichen aus t ab Index j

`s[j]`	Liefert Referenz auf bit mit Index j als `bool`-Wert
`s.test(j)`	Liefert Wert von bit mit Index j als `bool`
`s.size()`	Liefert N
`s.count()`	Liefert Anzahl von bits aus s mit Wert `true`
`s.any()` `s.all()` `s.none()`	Liefert `bool`-Wert ob mindestens eins bzw. alle bzw. keins der bits aus s den Wert `true` hat
`s.set(j, b)`	Setzt bit mit Index j auf Wert b
`s.set()` `s.set(j)`	Setzt alle bits bzw. bit mit Index j auf Wert `1`
`s.reset()` `s.reset(j)`	Setzt alle bits bzw. bit mit Index j auf Wert `0`
`s.flip()` `s.flip(j)`	Negiert alle bits bzw. bit mit Index j (`0` $\mapsto$ `1`, `1` $\mapsto$ `0`)
`s == s'` `s != s'`	Vergleicht s und s' auf Gleichheit
`~s` `s & s'` `s | s'` `s ^ s'` `s << j` `s >> j`	Bit-weise logische Operatoren
`s &= s'` `s |= s'` `s ^= s'` `s <<= j` `s >>= j`	Bit-weise logische Zuweisungsoperatoren
`s.to_ulong()`	Liefert einen Wert vom Typ `unsigned long` der dem Inhalt von s entspricht
`s.to_string()` `i >> s` `o << s`	Konvertierung von Zeichenketten bestehend nur aus den Zeichen `0` und `1` zu bzw. von s vom Typ `bitset<N>`

Beispiel (Sieb des Eratosthenes). Wir implementieren ein Hauptprogramm, dass das Sieb des Eratosthenes verwendet um Primzahlen kleiner als 50 zu finden und auszugeben. Wir verwenden hierbei ein Objekt vom Typ `bitset<50>` um für Zahlen kleiner 50 zu speichern ob diese nach aktuellem Stand prim sind oder nicht.

primes_bitset.cpp

```cpp
#include <bitset>
#include <iostream>

using namespace std;

const unsigned int n = 50;

int main() {
  bitset<n> primes;
```

```
  primes.set();
  primes.reset(0); primes.reset(1);

  for (unsigned int k = 2; k*k < n; k++)
    if (primes[k])
      for (unsigned int i = k; i*k < n; i++)
        primes.reset(i*k);

  for (unsigned int k = 0; k < n; k++)
    if (primes[k])
      cout << k << " ";
  cout << endl;

  return 0;
}
```

```
2 3 5 7 11 13 17 19 23 29 31 37 41 43 47
```

Endliche Abbildungen in der STL (`map`)

Objekte vom Datentyp `map<`K`, `V`>` speichern beliebig viele Paare vom Typ `pair<const `K`, `V`>` *ohne Duplikate* bzgl. der ersten Komponente der Paare (also keine Duplikate in Werten vom Typ K, Duplikate in den Werten vom Typ V sind erlaubt). Die Daten werden in einer geeigneten Baumstruktur für effizienten Zugriff anhand der Indizes vom Typ K gespeichert. Um die interne Baumstruktur bilden zu können muss `operator<` geeignet überladen sein für K.

Semantisch modelliert ein Wert vom Typ `map<`K`, `V`>` eine Abbildung $K' \to V$ mit $K' \subseteq K$ endlich.

Im Folgenden bezeichnen K, V Datentypen, m, m' Variablen vom Typ `map<`K`, `V`>`, k, $k_0, \ldots, k_{n-1}$ Variablen vom Typ K, v, $v_0, \ldots, v_{n-1}$ Variablen vom Typ V und p eine Variable vom Typ `pair<`K`, `V`>`. Zudem bezeichnen i, i' Iteratoren über m.

`map<`K`, `V`>` m	Vereinbart Variable m vom Typ `map<`K`, `V`>` leer
`map<`K`, `V`>` m`{{`k_0`,`v_0`}, ..., {`k_{n-1}`,`v_{n-1}`}}`	Vereinbart Variable m vom Typ `map<`K`, `V`>` mit m`[`k_i`]=`v_i für $0 \le i < n$
m`[`k`]`	Liefert Referenz auf Wert zum Index k. Falls k noch nicht existiert in m wird der Wert zum Index k zunächst mit Standardkonstruktor initialisiert.
m `==` m' m `!=` m' m `>` m' m `<` m' m `<=` m' m `>=` m'	Vergleicht m und m' lexikographisch
m`.size()`	Liefert Anzahl der Elemente in m
m`.clear()`	Löscht alle Elemente aus m
m`.empty()`	Liefert `true` g.d.w m leer ist
m`.count(`k`)`	Liefert `1` falls k Element zu einem Index enthält, der äquivalent ist zu k, sonst `0`

`m.insert(p)`	Fügt Index-Wert-Paar p ein in m. Liefert Paar aus Iterator auf Position von erster Komponente von p in m (bzw. Position mit Index äquivalent zu erster Komponente von p) und Wert vom Typ `bool` der angibt ob Einfügen stattgefunden hat (also kein Index äquivalent zur ersten Komponente von p bereits vorhanden war in m)
`m.erase(k)`	Löscht Index-Wert-Paar aus m mit Index äquivalent zu k, falls vorhanden. Liefert Anzahl gelöschter Index-Wert-Paare (`0` oder `1`).

`map<K, V>::iterator i`
`map<K, V>::const_iterator i`
`map<K, V>::reverse_iterator i`
`map<K, V>::const_reverse_iterator i`
	Vereinbart Variable i als Vorwärts-, Konstanten-, Rückwärts- bzw. Rückwärts-Konstanten-Iterator

`m.begin()` `m.end()` `m.rbegin()` `m.rend()`	Liefert Vorwärts-Iterator über m zum ersten bzw. *nach* dem letzten Index-Wert-Paar bzw. Rückwarts-Iterator über m zum letzten bzw. *vor* dem ersten Index-Wert-Paar
`++i` `i++` `--i` `i--`	Verschiebt Iterator i auf nächste bzw. vorherige Position in Durchlaufrichtung
`*i`	Referenz zu Index-Wert-Paar an aktueller Position von i
`i->first` `i->second`	Referenz zu Index bzw. Wert an aktueller Position von i
`m.find(k)`	Liefert Iterator zur Position mit Index äquivalent zu k in m falls vorhanden, sonst `m.end()`
`m'.insert(i, i')`	Fügt Kopien der Index-Werte-Paare aus m im Bereich $[i, i')$ ein in m'
`m.erase(i)` `m.erase(i, i')`	Löscht Index-Wert-Paar an aktueller Position von i bzw. im Bereich $[i, i')$ aus m. Liefert Iterator auf Index-Wert-Paar das direkt auf letztes nun gelöschtes Index-Wert-Paar gefolgt hat.

Beispiel. Wir implementieren ein Hauptprogramm zum Einlesen von Paaren bestehend aus dem Namen eines Kursteilnehmers und einer Anzahl von erreichten Übungspunkten. Die Namen fungieren als Indizes eines Objekts vom Typ `map<string, int>`, `points`. Zunächst wird `points` vorbelegt mit den Namen von Teilnehmern, die aus einer Datei (`teilnehmer.txt`) eingelesen werden. Im Anschluss wird vom Benutzer immer wieder die Eingabe eines Namens und einer Punktezahl erwartet. Falls unter diesem Namen bereits ein Eintrag in `points` existiert werden die Punkte addiert, ansonsten wird der Benutzer gefragt ob ein neuer Eintrag angelegt werden soll. Am Ende werden die eingegebenen Daten dann ausgegeben, wegen der internen Struktur von `map` in nach Namen sortierter Reihenfolge.

input_exc.cpp

```cpp
#include <map>
#include <string>
#include <iostream>
#include <iomanip>
#include <fstream>

using namespace std;
```

```cpp
int main(int argc, char* argv[]) {
  map<string, int> points;

  ifstream names(argv[1]);
  string s;
  while (names >> s) points[s];

  cout << "Teilnehmer  Punkte:" << endl;
  int n;
  while (cin >> s >> n) {
    if (points.find(s) != points.end())
      points[s] += n;
    else {
      char c;
      cout << "Neuer Teilnehmer (j/N)? "; cin >> c;
      if (c == 'j') points.insert(pair<string, int>{s, n});
    }
  }

  for (pair<string, int> entry: points)
    cout << setw(10) << entry.first
         << setw(3) << entry.second
         << endl;

  return 0;
}
```

teilnehmer.txt

```
Meyer
Mueller
Moser
Maier
```

```
Teilnehmer  Punkte:
Moser 3
Mueller 4
Maurer 2
Neuer Teilnehmer (j/N)? j
Moser 5
     Maier  0
    Maurer  2
     Meyer  0
     Moser  8
   Mueller  4
```

Beispiel. Wir implementieren ein Hauptprogramm um die häufigsten Worte im Inhalt einer gegebenen Datei zu bestimmen. Es werden zunächst alle (durch insb. Leerzeichen oder Zeilenumbrüche getrennten) Worte aus der als Kommandozeilenparameter gegebenen Datei eingelesen und in einem Objekt vom Typ `map<string, int>`, `freq`, mit der Häufigkeit ihres Vorkommens assoziiert. Durch die Verwendung von `operator[]` wird, sollte das Wort in `freq` noch nicht vorkommen, den Eintrag für diesen Index automatisch mit `0` initialisiert. Es werden dann die Index-Werte-Paare in einen Vektor kopiert und die Funktion `stable_sort` aus `<algorithm>` verwendet um diesen Vektor zu sortieren. Hierbei wird eine eigene Vergleichsfunktion für die Elemente des

Vektors (vom Typ `pair<string, int>`) verwendet die nur den zweiten Eintrag der Paare (hier die Häufigkeit des Vorkommens) betrachtet. Es werden dann die ersten 10 Einträge des derart sortierten Vektors ausgegeben.

freq.cpp

```cpp
#include <map>
#include <string>
#include <iostream>
#include <iomanip>
#include <fstream>
#include <vector>
#include <algorithm>

using namespace std;

bool ordnung(const pair<string, int>& a,
  const pair<string, int>& b) {
    return a.second > b.second;
}

int main(int argc, char* argv[]) {
  map<string, int> freq;
  int nwort = 0;

  ifstream ein(argv[1]);
  string wort;
  while (ein >> wort) {
    nwort++;
    freq[wort]++;
  }

  vector<pair<string, int>> vektor(freq.begin(), freq.end());
  stable_sort(vektor.begin(), vektor.end(), ordnung);

  int count = 0;
  for (auto entry: vektor) {
    cout << setw(10) << entry.first
         << setw(5) << entry.second
         << endl;
    if (++count >= 10) break;
  }

  cout << endl << "Wortzahl: " << setw(8) << nwort << endl;

  return 0;
}
```

Wir rufen das Programm auf mit dem Namen der folgenden Datei als Konsolenparameter.

heron_kurz.cpp

```cpp
int main ( )
{
  double a , b , c , s , F ;
  cout << "a b c: " ;
  cin >> a >> b >> c ;
```

```
  s = ( a + b + c ) / 2.0 ;
  F = sqrt ( s * ( s - a ) * ( s - b ) * ( s - c ) ) ;
  cout << "F = " << F << endl ;
  return 0 ;
}
```

```
          ;    7
          (    6
          )    6
          s    6
          b    5
          ,    4
         <<    4
          a    4
          c    4
          *    3

Wortzahl:      82
```