Deques "decks" [double ended queue] (#include <deque>)

- nach zwei Seiten dynamisches Array eines beliebigen Typs mit wahlfreiem Zugriff, unsortiert
- alle Algorithmen sind anwendbar (RandomAccessIterator)
- sehr gutes Zeitverhalten beim Löschen und Einfügen am Anfang und am Ende
- ansonsten ist jedes Löschen/Einfügen mit dem Verschieben (Zuweisen) von Elementen verbunden ! (--> schlechtes Zeitverhalten)
- Deques werden i.allg. in mehreren Speicherblöcken (anders als Vektoren) automatisch verwaltet:
  - es gibt keine Vorreservierung, implizite Reallokierung
  - bei jedem Einfügen werden **alle Verweise potentiell ungültig**
  - wahlfreier Zugriff ist zwar möglich aber langsamer als bei Vektoren

Ausdruck	Bedeutung
`wie vector`	Erzeugung ...
`keine weiteren`	Nicht verändernde Operationen ...
`wie vector`	Zuweisung ...
`wie vector auch at und []`	Zugriffe ...
`wie vector +`	Einfügen / Löschen ...
`m.push_front (e)`	Kopie von e vorn anhängen
`m.pop_front ()`	löscht erstes Element (gibt nichts zurück)

Listen (#include <list>)

- doppelt verkettete Liste eines beliebigen Typs ohne wahlfreien Zugriff (man muss sich "durchhangeln")
- unsortiert
- nicht alle Algorithmen sind anwendbar (BidirektionalIterator)
- gutes Zeitverhalten beim Löschen und Einfügen an beliebiger Stelle !
- Verweise auf Elemente bleiben dabei gültig !!

Ausdruck	Bedeutung
`wie deque`	Erzeugung ...
`keine weiteren (nur diese)`	Nicht verändernde Operationen ...
`wie deque`	Zuweisung ...
`wie deque ohne at und []`	Zugriffe ...
`wie deque +`	Einfügen / Löschen ...
`m.remove (e)`	löscht alle! Elemente mit dem Wert e
`m.remove_if (op)`	löscht alle! Elemente für die op(e) gilt
`m.erase (pos)`	löscht Element bei pos und liefert Position des Folgeelementes
`m.erase (f, l)`	löscht Elemente der Bereichs [f,l) und liefert Position des Folgeelementes

Ausdruck	Bedeutung
`m.unique ()`	kollabiert Folgen von Elementen mit gleichem Wert
`m.unique (op)`	kollabiert Folgen von Elementen mit op(e1)==op(e2)
`m1.splice (pos,m2)`	verschiebt alle Elemente von m2 nach m1 vor die Position pos
`m1.splice (pos,m2,m2pos)`	verschiebt das Element von m2 an der Position m2pos nach m1 vor die Position pos (m1, m2 identisch ist erlaubt)
`m1.splice (pos,m2,m2f,m2l)`	verschiebt alle Elemente von m2 aus dem Bereich [m2f, m2l) nach m1 vor die Position pos (m1, m2 identisch ist

Ausdruck	Bedeutung
`m.sort ()`	sortiert nach <
`m.sort (op)`	sortiert nach op
`m1.merge (m2)`	mischt sortiertes m2 in sortiertes m1 ein
`m1.merge (m2,op)`	mischt sortiertes m2 in sortiertes m1 ein, sortiert dabei nach op
`m.reverse ()`	kehrt die Reihenfolge der Elemente um

Mengen und - Multimengen (#include <set>)

- Mengencontainer mit automatischer Sortierung der Elemente
- `set` - jedes Element kommt höchstens einmal vor
- `multiset` - Elemente können mehrfach enthalten sein (Bags)
- wegen der automatischen Sortierung muss für den Elementtyp der Operator `<` definiert sein ! Dieser legt auch die Gleichheitsrelation fest: zwei Elemente a und b sind gleich, wenn weder a<b noch b<a gilt !
- man kann bei der Instantiierung von Mengentemplates auch ein anderes Ordnungskriterium angeben:

```
namespace std {
     template < class T,
                class Compare = less<T>,
                class Allocator = allocator<T> >
     class set; /* dito multiset */          }
```

- `set <int, greater<int> >` absteigend_sortierte_Mengen

- `greater<T>` und `less<T>` sind sog. function objects (`operator()` ist definiert) `#include <functional>`

```
template <class Arg, class Result>
  struct unary_function {
    typedef Arg    argument_type;
    typedef Result result_type;
  };
template <class Arg1, class Arg2, class Result>
  struct binary_function {
    typedef Arg1   first_argument_type;
    typedef Arg2   second_argument_type;
    typedef Result result_type;
  };
```

Funktoren

```
template <class T> struct greater : public
  binary_function<T,T,bool> {
  bool operator()(const T& x, const T& y) const
  { return x > y; }
};
```

Aufruf	Operation
`negate<T>()(p)`	- p
`plus<T>()(p1, p2)`	p1 + p2
`minus<T>()(p1, p2)`	p1 - p2
`multiplies<T>()(p1, p2)` `[depricated] times<T>()`	p1 * p2
`divides<T>()(p1, p2)`	p1 / p2
`modulus<T>()(p1, p2)`	p1 % p2

Funktoren

Aufruf	Operation
`equal_to<T>()(p1, p2)`	p1 == p2
`not_equal_to<T>()(p1, p2)`	p1 != p2
`less<T>()(p1, p2)`	p1 < p2
`greater<T>()(p1, p2)`	p1 > p2
`less_equal<T>()(p1, p2)`	p1 <= p2
`greater_equal<T>()(p1, p2)`	p1 >= p2
`logical_not<T>()(p)`	! p
`logical_and<T>()(p1, p2)`	p1 && p2
`logical_or<T>()(p1, p2)`	p1 || p2

Funktoradaptoren

Aufruf	Operation
`bind1st(op, wert)`	op(wert, param)
`bind2nd(op, wert)`	op(param, wert)
`not1(op)`	!op(param)
`not2(op)`	!op(param1, param2)

Beispiel: alle geraden Zahlen aus einer Liste entfernen

```
liste.remove_if (not1(bind2nd(modulus<int>(), 2)));
```

Mengen und - Multimengen (#include <set>)

- Implementierung typischerweise als balancierte Binärbäume (red-black-tree)
- sehr gutes Zeitverhalten beim Suchen, gutes beim Löschen und Einfügen an allen Stellen !
- die Sortierung hat zur Konsequenz, dass man Elemente nicht ändern kann, realisiert dadurch, dass alle Iteratoren Zugriffe auf `const`-Objekte bereitstellen (Wert ändern: alten Wert aufsuchen und löschen, neuen Wert einfügen)
- beim Einfügen einzelner Elemente unterscheiden sich `set` und `multiset` in ihren Rückgabewerten:

`multiset`	Position des neuen (eingefügten) Elements
`set`	gibt ein `pair<iterator,bool> p` zurück: `p.second` gibt an, ob wirklich eingefügt wurde, `p.first` ist die Position des eingefügten bzw. bereits vorhanden Elements

Ausdruck	Bedeutung
`m.count (e)`	Anzahl der Elemente mit Wert e `set`: 0..1; `multiset` 0..n
`m.find (e)`	liefert die Position des ersten Auftretens von e oder `end()`
`m.lower_bound (e)`	erste Position an der e eingefügt werden könnte (das erste Element >= e)
`m.upper_bound (e)`	letzte Position an der e eingefügt werden könnte (das erste Element > elem)
`m.equal_range (e)`	erste und letzte Position zum Einfügen gibt ein `pair<const_iterator,`

Ausdruck	Bedeutung
`m.insert (e)`	fügt Element ein und liefert Position des neuen Elements, bzw. ob es geklappt hat
`m.insert (pos,e)`	fügt Element ein und liefert Position des neuen Elements, bzw. ob es geklappt hat
`m.insert (f, l)`	Elemente von [f, l) einfügen (void)
`m.erase (e)`	löscht alle Auftreten von e, return Anzahl der entfernten Elemente
`m.erase (pos)`	löscht Element bei pos
`m.erase (f, l)`	löscht Bereich [f, l) liefert Position des Folgeelements

Maps und - Multimaps (#include <map>)

- Mengencontainer für Schlüssel/Wert- Paare mit automatischer Sortierung anhand der Schlüssel (dictionaries)

```
typedef pair<const Key, T> value_type; // in [multi]map
```

- **map** jeder Schlüsselwert kommt höchstens einmal vor
- **multimap** Schlüsselwerte können mehrfach enthalten sein
- wegen der automatischen Sortierung muss für den Schlüsseltyp der Operator < definiert sein ! dieser legt auch die Gleichheitsrelation fest: zwei Schlüssel a und b sind gleich, wenn weder a<b noch b<a gilt !

- die Sortierung hat zur Konsequenz, dass man Schlüssel nicht ändern kann, realisiert dadurch, dass alle Schlüssel `const`-Objekte sind, der Wert zu einem Schlüssel kann geändert werden !

- man kann bei der Instantiierung von `map`-Templates auch ein anderes Ordnungskriterium angeben

```
namespace std {
  template <
    class Key,
    class T,
    class Compare = less<Key>,
    class Allocator = allocator<pair<const Key, T> >
  >
  class map; // dito multimap
}
```

Ausdruck	Bedeutung
`wie set (count ... equal_range)`	e ist ein Schlüsselwert (`Key`)
`wie set (insert ... erase)`	e ist vom Typ `[multi] map<Key,T>::value_type`
`m[key]` **(nur für map !)**	liefert eine Referenz für den Wert des Elements zu key, fügt das Element dabei

– beim Einfügen einzelner Elemente unterscheiden sich `map` und `multimap` in ihren Rückgabewerten:

`multimap` Position des neuen (eingefügten) Elements

`map` gibt ein `pair<iterator,bool> p` zurück:
`p.second` gibt an, ob wirklich eingefügt wurde,
`p.first` ist die Position des eingefügten bzw. bereits vorhanden Elements

3. Generische Programmierung in C++

```
#include <iostream>
#include <string>
#include <map>

using namespace std ;

int main ( ) {
  string buf ;
  map < string , int > m ;
  while ( cin >> buf ) m [ buf ] ++ ;
  multimap < int , string > n ;
  for ( map < string , int > :: iterator p = m . begin ( ) ;
                          p !=  m . end ( ) ; ++ p )
    n . insert ( multimap < int , string > :: value_type ( p -> second , p ->
    first ) ) ;
  for ( multimap < int , string > :: iterator p = n . begin ( ) ;
                          p !=  n . end ( ) ; ++ p )
    cout <<  p -> first  << "\t" << p -> second << endl ;
}
```

```
$ wc < wc.cpp
1       "\t"
1       <iostream>
…..
6       string
10      (
10      )
10      p
11      ;
```

Input - & Output – Iteratoren

```
#include <string>
#include <iterator>
#include <algorithm>
#include <vector>
#include <iostream>
#include <sstream>

int main()
{
    std::istringstream s("I need your help!");

    std::vector<std::string> v( (std::istream_iterator<std::string>(s)) ,
                                std::istream_iterator<std::string>());
    std::copy(v.begin(), v.end(),
              std::ostream_iterator<std::string>(std::cout, "\n"));
}
```

```
I
need
your
help!
```

Scott Meyers „Effective STL" (Item 6:) Be alert for C++'s most vexing parse

	vector	deque	list	set	multiset	map	multimap
interne Datenstruktur	dynamisches Array	Menge von Arrays	doppelt verkettete Liste	Binärbaum	Binärbaum	Binärbaum	Binärbaum
Elemente-Art	Wert	Wert	Wert	Wert	Wert	Wertepaar	Wertepaar
Duplikate erlaubt	ja	ja	ja	nein	ja	nein (Schlüssel)	ja (Schlüssel)
wahlfreier Zugriff	ja	ja	nein	nein	nein	über Schlüssel	nein
Iterator-Kategorie	Random-Access	Random-Access	Bidirectional	Bidirectional (Wert konstant)	Bidirectional (Wert konstant)	Bidirectional (Schlüssel konstant)	Bidirectional (Schlüssel konstant)
Suchen/Finden von Elementen	langsam	langsam	sehr langsam	sehr schnell	sehr schnell	sehr schnell für Schlüssel	sehr schnell für Schlüssel
Einfügen/Löschen schnell	am Ende	am Anfang und am Ende	überall konstant	überall logarithmisch	überall logarithmisch	überall logarithmisch	überall logarithmisch
Verweise werden ungültig	bei Reallokierung	potenziell immer	nein	nein	nein	nein	nein
Speicher wird freigegeben	nie	manchmal	immer	immer	immer	immer	immer
Speicherreservierung möglich	ja	nein	-	-	-	-	-

Container-Adaptoren

neben den (primären) Containern gibt es einige sog. Container-Adaptoren, es handelt sich dabei um Anpassungen der Container für spezielle Anwendungen

Queues (`#include <queue>`)

FIFO-Warteschlangen (auch mittels `list` instantiierbar)

```
namespace std {
  template < class T,
             class Container = deque<T>
  >
  class queue;
}
```

Priority Queues (#include <queue>)

Warteschlangen mit Prioritäten (auch mittels `deque` instantiierbar)

```
namespace std {
  template < class T, class Container = vector<T>,
             class Compare = less<typename
Container::value_type>
  >
  class priority_queue;
}
```

Stacks (#include <stack>)

Kellerspeicher (auch mittels `list` und `vector` instantiierbar)

```
namespace std {
   template < class T, class Container = deque<T> >
   class stack;
}
```

Strings (#include <string>) (vgl. z.B. Josuttis, Kapitel 10, S.357 ff.)

```
namespace std {
  template < class charT,
     class traits = char_traits<charT>
     class allocator = allocator <charT> >
  class basic_string;
  // noch nicht auf Zeichentyp festgelegt
  typedef basic_string<char>    string;  // ASCII
  typedef basic_string<wchar_t> wstring; // Unicode
}
```

mit Einführung von `string` wurde auch die `iostream`-Bibliothek erheblich überarbeitet, um mit strings zusammenarbeiten zu können, ohne dass sich die Nutzerschnittstelle wesentlich verändert hat, ggf. ist wichtig

```
typedef basic_ostream<char, char_traits<char> > ostream;
```