Themen

- STL
  - Standard Template Library
    - Datenstrukturen
      - Container
      - Iteratoren
    - Algorithmen und Funktoren (kein Klausurthema)

STL - Einleitung

- Template Bibliothek mit Schablonen für Datenstrukturen, Iteratoren, Algorithmen, uvm.
- Vereinfacht zeitraubende Arbeiten (z.B. Arrays wachsen automatisch)
- Teil der C++ ISO/ANSi Standards

STL nutzen !!!

- **Effizient** – Algorithmen und Datenstrukturen sind effizient implementiert
- **Schnell** – komplexe Datenstrukturen sind schnell implementiert
- **Sauber** – hohe Konsistenz der Datenstrukturen
- **Standardkonform und portabel**
- **Gut getestet**
- **Gut dokumentiert**

Begriffe

- **Container**:
  - Enthält andere Objekte
  - Speichert Menge von Werten gleichen Typs
  - z.B. **`list`, `vector`, `set`, `map`, ...**
- **Iteratoren**:
  - Navigieren in einem Container
  - z.B. **`random`, `forward`, ...**
- **Algorithmen**:
  - Verändern oder durchsuchen den Inhalt eines beliebigen Objekts/Containers
  - z.B. **`sort()`, `find()`, `reverse()`,`replace()`**, ...

Beispiel

```
#include <iostream>
#include <vector>
int main() {
   // erzeuge Vektor mit 5 Elementen vom Typ int
   std::vector<int> myV(5); // wird initialisiert
   myV[0]=1;
   myV[2]=5;

   std::cout << "Elemente des Vektors: ";
   for(int i=0; i<5; i++) {
      cout << myV[i] << " ";
   }
}
```

Ausgabe: 1 0 5 0 0

Container-Typen

- **Sequentiell**: lineare Speicherung, keine Sortierung
  - Dynamische Felder – **`vector`**
  - Dynamisches Feld, dass auch nach vorne wachsen kann – **`double ended queue`**
  - Doppelt verkettete Liste – **`list`**
  - Eingeschränkte Container – **`queue, stack`**
- **Assoziativ**: jedes Element besitzt Schlüssel, Sortierung nach Schlüsseln
  - Menge – **`set`**
  - Menge mit Mehrfachvorkommen – **`multiset`**
  - Zuordung von Daten zu Schlüsseln – **`map`**
  - Nicht-eindeutige Zuordnung - **`multimap`**

Container verwenden

- Entsprechende Include-Datei einbinden
- Namen sind im Namensraum **`std`** definiert
- Template mit gewünschtem Typ instanziieren
- Wie “normales” Objekt verwenden, z.B. vorhandene Methoden aufrufen

Container - Eigenschaften

- Methoden in allen Containern (Auswahl):
  - **`STLContainer<T>()`** - initialisieren
  - **`bool empty()`** - Test, ob Container leer
  - **`size_type size()`** - Anzahl der Elemente
  - **`bool operator==(const STLContainer &other)`** – auf Gleichheit prüfen
  - **`void clear()`** - alle Elemente löschen

Container – Eigenschaften (2)

- Container **speichern** nur **Kopie** des eingefügten Elements
  - Eigene Speicherverwaltung
  - Wenn **nicht** kopiert werden soll, **Zeiger** verwenden
- **Vergleichsoperatoren**:
  - `==, !=, <,<=, >, >=`
  - Beide Container müssen **vom selben Typ** sein
  - Zwei Container sind gleich, wenn ihre Elemente gleich und in gleicher Reichenfolge sind
  - z.B. `<` : lexikographischer Vergleich

Sequentielle Container
`vector, deque, list, queue, stack`

- Methoden:
  - **`STLContainer<T>(size_type n, const T &t)`** – Container für Datentypen `T` mit `n` Kopien von `t` initialisieren
  - **`void push_front(const T &t)`** – Kopie von t am Anfang einfügen
  - **`void push_back(const T &t)`** – Kopie von t am Ende einfügen
  - **`void pop_front()`** - erstes Element entfernen
  - **`void pop_back()`** - letztes Element entfernen
  - **`reference front()`** - Referenz auf erstes Element zurückliefern
  - **`reference back()`** - Referenz auf letztes Element
  - **`reference operator[](size_type n)`** – Zugriff auf `n`-tes Element

vector

- **`#include <vector>`**
- Schnelles Einfügen und Entfernen nur am Ende, Suchen langsam
- Feld dynamischer Länge
  - wächst am Ende
    - **`push_front()`** und **`pop_front()`** nicht vorhanden
- Intern als Array repräsentiert
  - Zugriff über `[]`-Operator möglich
    - Beliebiger Zugriff auf Elemente
- Spezielle Vektoren:
  - **`vector<bool>`** ist platzoptimiert
    - **`void flip()`** - Bitkompliment aller Einträge

vector - Beispiel

```
#include <iostream>
#include <vector>
int main () {
  std::vector<int> v(10);
  unsigned int i;

  for (i=0; i<v.size(); i++) {
    v.at(i)=i; // Werte zuweisen
   }
  for (i=0; i < v.size(); i++) { // Ausgabe
     std::cout << " " << v[i]; // identisch zu at()
     std::cout << " " << v.at(i); // Unterschied!
   }
  return 0;
}
```

deque

- **`#include <deque>`**
- Warteschlange mit **zwei Enden** (double ended queue)
- Ähnlich Vector, aber nach hinten als auch auch nach vorn effizient erweiterbar
- Schnelles Einfügen an beiden Enden
- Intern durch Blocks repräsentiert, deren erster vorn und deren letzter nach hinten wächst
- Beliebiger Zugriff auf Elemente

list

- **`#include <list>`**
- Doppelt verkette Liste
- Überall schnelles Einfügen
- Kein `[]`-operator, kein wahlfreier Zugriff
- Sonderfunktionen:
  - **`unique(funkt)`** – entfernt Duplikate aufeinander folgender Elemente (default: Gleichheit)
  - **`sort(funkt)`** – Elemente nach `funkt` sortieren (default: < Aufsteigend)
  - **`liste1.merge(liste2, funkt)`** – Elemente beider Listen einfügen, so dass Sortierung gemäß `funkt` beibehalten wird

queue

- **`#include <queue>`**
- **FIFO**-Zugriff auf Elemente
- Intern als **`deque`** realisiert
- Einfügen am **Ende**: **`void push()`**
- Entfernen am **Anfang**: **`void pop()`**
- **Keine `pop_/push_back()/front()`**
  - Um queue zu bearbeiten **`front()`** und **`pop()`** kombinieren

stack

- `#include <stack>`
- **LIFO**-Zugriff auf Elemente
- Intern als `deque` realisiert
- Einfügen am **Ende**: `void push()`
- Entfernen am **Ende**: `void pop()`
- Zugriff auf nächstes Element:
  - `Reference top()`
  - **Keine** `front/back()`

Assoziative Container

`set, multiset, map, multimap`

- Verallgemeinerung von Sequenzen
- Sequenzen werden durch Integerwerte indiziert, assoziative durch beliebige Typen
- Elemente automatisch sortiert
- Sortierkriterium per Konstruktor übergeben (siehe Funktoren)

set, multiset

- **`Set`** – **Menge (**jeder Wert höchstens **einmal)**
- **`Multiset`** – Werte treten **mehrfach** auf
- Methoden:
  - **`set<T>(op), multiset<T>(op)`** – op default: Funktor **less<T>** (siehe Funktoren)
  - **`size_type count(const T &k)`** – Anzahl Elemente mit Wert == k
  - **`insert(const T &k)`** – Kopie von Element k einfügen
  - **`size_type erase(const T &k)`** – Elemente mit Wert k löschen (gibt Anzahl der gelöschten Elemente zurück)

set

- `#include <set>`
- Sortierter Container, der nur den Schlüssel enthält
- Intern als balancierter Binärbaum
  - Suchen sehr effizient
  - Elemente müssen **sortierbar** sein (aufsteigend)

Set - Beispiel

```
#include <iostream>
using namespace std;
#include <set>
int main() {
   set<int> Menge;
   int s;
   cout << "Set: Groesse: ";
   for(int j=4;j>=0;j--) {
      Menge.insert(j);
      Menge.insert(5);
      cout << Menge.size(); // Ausgabe: 2 3 4 5 6
   }
   Menge.erase(5); // Menge: 2 3 4 6
}
```

map, multimap

- **`map`** – Menge von Schlüssel-Wert-Paaren (jeder Schlüssel nur einmal vorhanden)
- **`multimap`** (Schlüssel mehrfach)
- Methoden:
  - **`map<Keytype, Valuetype>(op)`**
  - **`multimap<keytype, Valuetype>(op)`**
  - **`size_type count(const Keytype &k)`**
  - **`size_type erase(const Keytype &k)`**
  - **`Valuetype &operator[](const Keytype &k)`** – Zugriff auf Wert, der unter Schlüssel k gespeichert ist

map, multimap

- `#include <map>`
- `#include <multimap>`
- Effizientes Ablegen und Abfragen über Schlüssel
- Intern balancierter Binärbaum
- Schlüssel muss sortierbar sein
- Direktes Ändern der Schlüssel nicht erlaubt
- Falls Schlüssel nicht vorhanden, wird er hinzugefügt

Beispiel map

```
#include <map>

std::map<std::string, int> myMap;
std::string key1 = “alpha23”

// einfügen des Schlüssel-Werte-Paares
// (“alpha23”,1000)
myMap.insert(std::pair<std::string,int>
                    (key1,1000));

// oder
myMap[key1] = 1000;
```

Container und Datentypen

- Container können Typ X enthalten, wenn X folgende Methoden definiert:
  - `X::X();`
  - `X::X(const X&);`
  - `X& X::operator= (const X&);`
- Assoziative Container benötigen (sortieren)
  - `bool X::operator< (const X&);`

Iteratoren

- Einige Container besitzen keinen `[]`-operator
  - Zugriff auf Elemente über Iteratoren
- **Iteratoren**:
  - Objekt, das **wie Zeiger** benutzt werden kann, um Objekte eines STL-Container zu durchlaufen (ohne Wissen der internen Struktur)
  - Speichert **Position** eines Objekts
  - Mit * kann man auf **Wert** des Objekts an beschriebener Position zugreifen

Iteratoren wie Zeiger

- Zeigerarithmetik möglich
  - **`STLContainer<T>::iterator`** - Iterator-Datentyp für jeweiligen STL-Container
  - **`++myIterator`** – nächstes Element
  - **`--myIterator`** – vorheriges Element
  - **`*myIterator / myIterator->`** - Iterator derefenzieren
  - **`myIterator == otherIterator`** - Vergleich

Iteratoren und Container

- Werden von Containern bereit gestellt
  - Methoden, die Iteratoren zurückgeben oder als Parameter bekommen
    - **`myContainer.begin()`** - liefert Iterator, der auf das erste Element im Container zeigt
    - **`myContainer.end()`** - liefert Iterator, der auf das Ende des Containers zeigt (nicht letztes Element!)
    - **`myContainer.insert(iterator pos, T elem)`** – fügt Element an Stelle pos ein
    - **`myContainer.erase(iterator begIt, iterator endIt)`** – entfernt Elemente im Intervall `[begIt; endIt]`

Beispiel Iterator 1/3

```
#include <iostream>
#include <list>
using namespace std;

int main() {
   list<char> alphabet; // list Container mit Zeichen
   // Element einfügen: 'a' bis 'z'
   for (char c = 'a'; c <= 'z'; ++c) {
      alphabet.push_back (c);
   }
   // Ausgabe aller Zeichen über Iterator
   list<char>::iterator pos; // Iterator deklarieren
   for (pos = alphabet.begin(); pos != alphabet.end(); ++pos){
      cout << *pos << ' ';
   }
   cout << endl;
   return 0;
}
```

Beispiel Iterator 2/3

```
std::map<std::string, int> myMap;
std::string key1 = "alpha23";

// einfügen eines (Schlüssel, Wert)-Paares
// ("alpha23",1000)
myMap.insert(std::pair<std::string,int>
                    (key1,1000));

// Iterator für map
std::map<std::string, int>::iterator it;
//Suche nach Element in Map über den Schlüssel
it = myMap.find(key1);
if (it != myMap.end()) { // Element gefunden?
   int val = (*it).second; // Zugriff auf Wert
   std::string key = (*it).first }
```

Beispiel Iterator 3/3

```
#include <list>

std::list<std::string> myList;  // Liste
std::list<std::string>::iterator it; // Iterator

// einfügen der Werte
myList.push_back("abc"); // am Ende einfügen
myList.push_back("xyz"); // Liste: ("abc","xyz");

it = myList.begin();
++it; // it zeigt auf "xyz"
myList.insert(it,"ttt"); // Liste:("abc","ttt","xyz")

//Suche nach Element in Liste
it = myList.find("xyz");
if (it != myList.end()) // Element gefunden?
   int val = *it; // Zugriff auf Wert
```

Iteratoren - Hinweis

- Sobald Container verändert wird, Iterator-Wert ungültig und muss ggf. neu initialisiert werden (Ausnahme: `list`, wenn nicht das Element gelöscht wird, auf das der Iterator gerade zeigt)
- Iteratorgültigkeit wird nicht automatisch geprüft, wenn Container geändert wurde

Iterator-Typen

- Verschiedene Iteratoren haben verschiedene Fähigkeiten:
  - Input und Output Iterator
  - Forward Iterator
  - Bi-directional Iterator
  - Random Access Iterator
- Eigenschaften **hängen vom Container ab**
- Jeder Algorithmus in STL benötigt spezifischen Iterator

Input Iterator

- Eigenschaften:
  - Lesezugriff
  - Elementweise Vorwärts-Schritte
  - Rückgabe des aktuellen Element-Wertes
  - Ein Element kann nicht erneut gelesen werden
- Zum Einlesen von Daten oder Durchlaufen von Containern

Output Iterator

- Eigenschaften:
  - Schreibzugriff
  - Elementweise Vorwärts-Schritte
- Zum Schreiben von Output-Streams

Output Iterator

- Eigenschaften:
  - Schreibzugriff
  - Elementweise Vorwärts-Schritte
- Zum Schreiben von Output-Streams

```
//Einlesen von Zahlen von der Standardeingabe
vector<int> V;
copy(istream_iterator<int>(cin),
   istream_iterator<int>(), back_inserter(V));

// istream_iterator – Input - zum Einlesen
// back_inserter – Output Iterator - zum
// Schreiben in den Vektor
```

Andere Iteratoren

- Forward Iterator:
  - Kombiniert Funktionalität von Input- und Output Iteratoren
  - **`++iterator`**
- Bi-directional Iterator:
  - Wie Forward, kann aber auch rückwärts gehen
  - **`--iterator`**
  - **`z.B. list, map, set`**
- Random Access Iterator:
  - Wie bi-directional, aber kann beliebig auf Elemente zugreifen (**`z.B. vector`**)
  - **`it[n]`** – Zugriff auf Element **`it + n`**

STL-Algorithmen

- Funktionstemplates zur Manipulation von Containern, die mit entsprechenden Iteratoren instanziiert werden
- Anwendbar auf alle Container, die die benötigten Iteratoren zur Verfügung stellen
- **`find()`, `sort()`** und **`merge()`** stehen für alle Container zur Verfügung und sollten eher genutz werden als die allg. Algorithmen
- **`#include <algorithm>`**

Funktoren

- **Definition**:
  - Funktions-Objekte: Ein Objekt einer Klasse, dass sich wie eine Funktion aufrufen läßt
- **Implementierung**:
  - Klassen, die den ()-Operator überschreiben
- **Wozu**:
  - Um **Flexibilität** der Algorithmen zu erhöhen, Funktoren zur Parameterisierung verwenden
    - Instant einer Klasse kann zusätzliche Elemente haben

Funktoren - Beispiel

```
class Summe {
  public:
    unsigned operator() (unsigned z1,unsigned z2){
       return(z1 + z2);
    }
};

int main(){
  Summe sum;                  // Instanziierung
  unsigned a = 19;
  unsigned b = 19;
  unsigned c = sum(a, b);  // Aufruf
  return (static_cast<int> (c)); // c = 38
}
```

Funktoren – Beispiel 2

```
class ChangeFunktor() {
  public:
     unsigned maxValue;
     ChangeFunktor(unsigned max) {
        maxValue = max; }
     void operator() (unsigned &a){
       if (a > maxValue) {
          a = 0;  // Alle Werte über max werden 0
        }}};
int main() {
  std::vector<unsigned> contZahlen;  // STL-Vektor
   /* Container bitte hier füllen */
  std::for_each(contZahlen.begin(),
           contZahlen.end(), ChangeFunktor(50));
   /* Nun ist jeder Wert über 50 auf 0 gesetzt.*/
}
```

Funktoren in STL

- #include <functional>
  - `plus<T>` - wendet operator + an
  - `minus<T>` - wendet operator - an
  - `multiplies<T>` - wendet operator * an
  - `divides<T>` - wendet operator / an
  - `modulus<T>` - wendet operator % an

Funktoren in STL 2

- **Vergleichsfunktoren**:
  - `equal_to<T>` operator ==
  - `not_equal_to<T>` operator !=
  - `greater<T>` operator >
  - `greater_equal<T>` operator >=
  - `less<T>` operator <
  - `less_equal<T>` operator <=
- **Logikfunktoren**:
  - `logical_and<T>` operator &&
  - `logical_or<T>` operator ||
  - `logical_not<T>` operator !

Algorithmen in STL 1/2

- **Nicht-modifizierend:**
  - `count(beg, end, value)`
    - Zählt die Elemente mit dem Wert value im Bereich
  - `min/max_element(beg, end)`
    - Wert des minimalen/maximalen Elements
  - `find(beg, end, value) -> iterator`
    - Liefert Iterator auf erstes Element mit Wert value
  - `equal(beg, end, beg2)`
  - `equal(beg, end, beg2, CmpOp)`
    - Vergleicht zwei Sequenzen

Algorithmen in STL 2/3

- **Modifizierend:**
  - `for_each(beg, end, funktion/funktor)`
    - Funktion/Funktor verändert ein Element

```
void square(int& elem) {
   elem = elem * elem;
}

// Quadrierung alle Elemente in li
list<int> li; ...
for_each(li.begin(), li.end(), square);
```

Algorithmen in STL 3/4

- **Modifizierend:**
  - `transform(beg, end, dest, funktion/funktor)`
    - Funkt. liefert das veränderte Element als Returnwert

```
int square1(int elem) {
   return elem * elem;
}

transform(c.begin(), c.end(),
          d.begin(), square1);
```

Algorithmen in STL 4/4

- **Numerisch**: `#include <numeric>`
  - `partial_sum(beg, end, dest, Op)`
    - Liefert die Partialsummen des Bereichs zurück
  - `adjacent_difference(beg, end, dest, Op)`
    - Liefert die Differenzen aufeinanderfolgender Elemente
  - ....

Literatur

- Referenz mit Beispielen zum nachschauen: http://www.cplusplus.com/reference/
- Josuttis: The Standard Template Library (Gutes Nachschlagewerk zur C++ STL.)