Die Standard Template Library

Zu einem der wesentlichen positiven Merkmale und Entwicklungen von C++ zählt die Standardisierung nach ANSI und ISO. Dazu gehört auch die Festlegung der Standard Template Library (STL). In ihr sind eine Reihe von nützlichen Klassen und Funktionen festgelegt:

Container: Dies sind Objekte, die andere Objekte (Elemente) verwalten können. Sie repräsentieren verschiedene abstrakte Datenstrukturen und können mit Iteratoren die jeweiligen Elemente einfach bearbeiten.

Iterator: Dies ist eine Abstraktion (Verallgemeinerung) von Zeigern und dient dem Zugriff auf Container - Elemente und/ oder arrays und wird von Algorithmen benutzt.

Algorithmen: Sie sind containerunabhängig und können nur über Iteratoren auf die Datenstruktur des jeweiligen Containers zugreifen. Da Algorithmen mit jeder Art von Iteratoren zusammenarbeiten können, muss jeder Algorithmus in nur einer Form vorhanden sein - dadurch werden Inkonsistenzen ausgeschlossen und Verwaltungsprobleme minimiert.

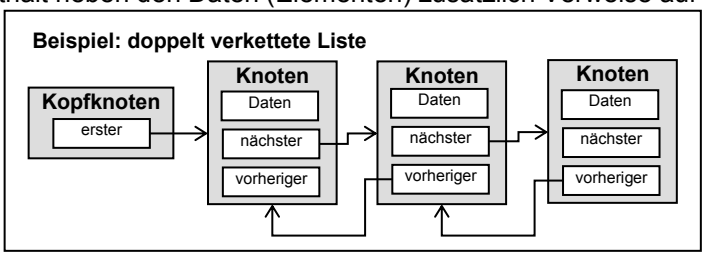
1. Arten von Containern

In der STL sind eine Reihe von Containern und Container – Adaptern (Container, welche wiederum Container enthalten können) vordefiniert. Container gliedert man dabei in verschiedene Gruppen auf.

1.1 Listen

Listen gehören zu den sequentiellen Containern das heißt die gespeicherten Elemente sind streng geordnet. Listen speichern ihre Daten in Knoten. Ein Knoten enthält neben den Daten (Elementen) zusätzlich Verweise auf weitere Knoten. Man unterscheidet zwischen:

- **einfach verketteten Listen**: Ein Knoten hält nur den Verweis auf den nächsten Knoten. Hier kann man den Iterator nur in einer Richtung zum nächsten Knoten bewegen und
- **doppelt verketteten Listen**: Ein Knoten enthält Verweise auf den vorherigen und den folgenden Knoten. Die doppelt verkettete Liste kann mit Iteratoren in beide Richtungen durchlaufen werden.



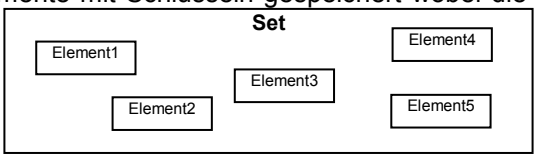
Beim Einfügen neuer Elemente sind nur die Verweise nächster und vorheriger anzupassen. Dadurch können Elemente sehr leicht hinzugefügt werden.

Beispiele aus der STL:

- **deque**: sollte benutzt werden, wenn häufiges Einfügen am Anfang und am Ende der Liste erwartet wird.
- **list**: ist auf das Einfügen von Daten in der Mitte des Containers optimiert.
- **vector**: sollte standardmäßig verwendet werden.

1.2 Sets

Sets zählen zu den assoziativen Containern. In ihnen werden Elemente mit Schlüsseln gespeichert wobei die Elemente selbst den Schlüssel darstellen. Sets unterstützen Abbildungen von Mengen in einem Computer. Das Einfügen neuer Elemente verläuft etwas langsamer als bei den Listen dafür sind diese Container auf das Suchen von Elementen über den Schlüssel optimiert.

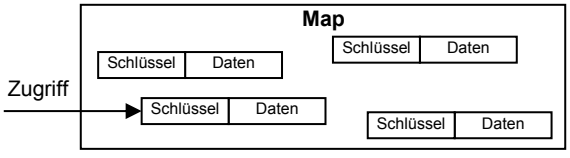


Beispiele aus der STL:

- **set**: beim set dürfen keine doppelten Elemente (Elemente mit dem gleichen Schlüssel) vorhanden sein.
- **multiset**: im multiset kann es gleiche Schlüssel geben, es erfolgt ein schneller Zugriff auf die Elemente über die Schlüssel.

1.3 Maps

Maps (auch als hash-tables bezeichnet) gehören zu den assoziativen Containern und arbeiten mit Schlüsseln (keys). Die Schlüssel und die dazugehörigen Daten können von unterschiedlichem Datentyp sein. Elemente einer Map setzen sich aus einem Schlüssel/ Daten Paar zusammen. Der schnelle Zugriff auf die Daten erfolgt über den Schlüssel.



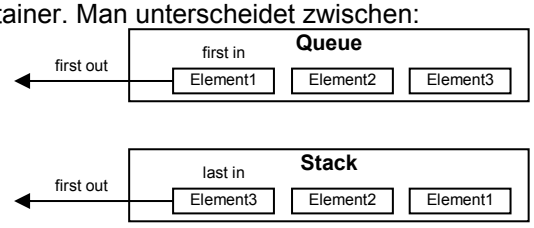
Beispiele aus der STL:

- **map**: ein Schlüssel kann nur auf ein enthaltenes Element der Map verweisen.
- **multimap**: ein Schlüssel kann auf mehrere Elemente verweisen.

1.4 Container - Adapter

Container Adapter der STL setzen auf andere sequentielle Container auf. Sie implementieren besondere Arten des Einfügens und Herausnehmens von Elementen aus einem Container. Man unterscheidet zwischen:

- **Queue**: Ein Queue arbeitet nach dem **fifo** Prinzip (first in first out). Elemente die zuerst abgelegt werden, können auch als erstes wieder aus dem Queue herausgenommen werden und
- **Stack**: Ein Stack (Stapelspeicher) funktioniert nach dem Prinzip last in first out (**lifo**). Elemente die zuletzt abgelegt werden, sind zuerst wieder zu entnehmen.



Beispiele aus der STL:

- **queue**: setzt auf deque oder list auf und arbeitet nach dem fifo Prinzip.
- **priority_queue**: Elemente werden nach einer Priorität sortiert abgelegt und immer das Element mit der höchsten Priorität zurückgeliefert. Er arbeitet mit deque und vector zusammen.
- **stack**: first in last out. stack benutzt die Container deque, list oder vector.

2. Die Verwendung von Algorithmen:

Algorithmen sind fertige Funktionstemplates, mit denen der Inhalt von Containern bzw. arrays bearbeitet werden kann. Zum Verständnis für die Funktionsweise von Algorithmen muss zunächst ein kleines Testprogramm mit zwei arrays geschrieben werden:

Um die Algorithmen nutzen zu können, ist die Headerdatei <algorithm> einzubinden.

☞ **Erzeugen Sie in main zwei arrays mit Zufallszahlen bzw. einem Wort!**

```
...
#include <algorithm>    // für Algorithmen
...
int aZahlen[12]={1, 4, 77, 45, 34, 22, 33, 45, 6, 67, 8, 9};
char aWort[12]  = "Algorithmen";
...
```

☞ **Erstellen Sie die Funktion void Ausgabe(int aArray[12]), die mit Hilfe einer for – Schleife den Inhalt des gesamten arrays auf dem Bildschirm ausgibt.**

- Zwischen den Zahlen soll dabei ein Leerzeichen ausgegeben werden!
- Am Ende der Funktion erfolgt ein Zeilenwechsel!

☞ **Testen Sie die Ausgabe für das int-array!**

Für den Test der Beispielalgorithmen ist es nützlich, auch eine Funktion zur Ausgabe des char arrays zu erstellen. Nun könnte man eine zweite Funktion für char deklarieren... Stop! Faule Programmierer beherrschen ja bereits die Fähigkeit ein Funktionstemplate zu entwerfen, dass int-arrays und char-arrays gleichermaßen auf dem Bildschirm präsentieren kann ☺ !

☞ **Erstellen Sie ein Funktionstemplate für die Funktion Ausgabe(...) und testen Sie dessen Funktion!**

Zunächst soll das Funktionstemplate

```
template<class RandomAccessIterator>
  void sort(RandomAccessIterator first, RandomAccessIterator last)
```

getestet werden.

☞ **Fügen Sie dazu folgenden Quelltext in die Hauptfunktion main( ) ein und verwenden Sie die Funktion Ausgabe( )!**

Da Iteratoren eine Verallgemeinerung von Zeigern darstellen, können sie in einem array genutzt werden. Als Parameter für die Funktion sort werden die Iteratoren first und last übergeben. Innerhalb des Bereichs first...last werden alle Elemente des arrays sortiert. Da Iteratoren mit Zeigern gleichzusetzen sind, werden mit dem Adressoperator & die Anfangsadresse des ersten gespeicherten Wertes im array (&aZahlen[0], und die Adresse nach dem letzten gespeicherten Wert (&aZahlen[12]) des arrays übergeben. Damit sortiert die Funktion sort(...) alle Werte im array.

```
sort(&aZahlen[0], &aZahlen[12]);
sort(&aWort[0], &aWort[12]);
```

Alle Algorithmen in der STL darzustellen würde den zeitlichen Rahmen des Unterrichts sprengen. Daher sollen folgend vier Beispiele zum Test vorgestellt werden.

☞ **Testen Sie die Funktionsweise der folgenden Algorithmen:**

template<class BidirectionalIterator> void **reverse**(BidirectionalIterator first, BidirectionalIterator last)	template<class ForwardIterator, class T> ForwardIterator **remove**(ForwardIterator first, ForwardIterator last, const T& value)
template<class ForwardIterator, class T> void **replace**(ForwardIterator first, ForwardIterator last, const T& old_value, const T& new_value)	template<class RandomAccessIterator> void **random_shuffle**(RandomAccessIterator first, RandomAccessIterator last)

3. Die Verwendung von Containern

Container sind Behälter, die Daten eines Typs speichern (vergleichbar mit einem Array). Sie passen ihren Speicherbedarf aber automatisch dem Inhalt an. Auf die Inhalte eines Containers kann nur über Iteratoren zugegriffen werden. Einen Iterator kann man sich dabei als Zeiger auf die Speicheradresse eines Elements des Containers vorstellen.

Am Beispiel des vector – Containers aus der STL soll der Umgang mit Containern verdeutlicht werden:

3.1 Erzeugen des Containers

Zum Testen der Funktionalität dient folgendes Programm, in dem verschiedene Werte vom Typ char in einem vector – Container gespeichert und bearbeitet werden:

Um einen vector – Container nutzen zu können, ist die Headerdatei <vector> einzubinden.

```
#include <vector>
...
```

In der Hauptfunktion main( ) erstellt man den vector – Container cont1, der Werte vom Typ char speichern kann. Über das Array test und cin wird ein Wort eingegeben.
Anschließend sollen alle Buchstaben des Wortes der Reihe nach in den Container geschrieben werden.

```
vector<char> cont1;
char test[20];
cout << "Bitte ein beliebiges Wort eingeben:";
cin >> test;
...
```

Mit der Funktion **push_back( )** kann ein Wert am Ende des Containers eingefügt werden. Dazu benutzt man den **Punktoperator** : cont1.push_back(...);

☞ **Erzeugen Sie eine while – Schleife, die alle Buchstaben aus „test“ in den Container einliest!**

- Tipp: Ein Zeichenarray (wie test) hat als letztes Zeichen am Ende des Wortes ein \0 ! Es sollten also alle Werte aus den Feldern eingelesen werden, solange ...

3.2 Arbeiten mit dem Container

Mit den Funktionen size_type size() const und size_type max_size() const können Sie nun die Anzahl der Elemente bzw. die maximale Anzahl der Elemente in cont1 anzeigen lassen.

☞ **Testen Sie die Funktionen size( ) und max_size( )!**

☞ **Zur Anzeige des Containerinhalts ist die Funktion Show( ) zu erstellen!**

Die Funktion Show gibt nichts zurück. Sie übernimmt aber eine Liste mit char Werten als Referenz.
Um auf den Inhalt des Containers zugreifen zu können, erzeugt man einen Iterator I. Dieser wird mit der Funktion begin( ) auf die Adresse des ersten Wertes im Container gesetzt. Danach erfolgt die Ausgabe in einer while – Schleife.

☞ **Erklären Sie die Funktion der Ausgabe in der while – Schleife!**

```
void Show(vector<char> &v)
{
   vector <char>::iterator I;
   I = v.begin( );                  // Iterator aufs erste
                                    // Element setzen

   while(I != v.end( ))             // Ausgabe bis
         cout << *I++;              // Ende erreicht ist

   cout << endl;
}
```

_____

_____

_____

3.3 Löschen des Containerinhalts

Um den Container wieder zu leeren verwenden Sie die Funktion void erase(iterator first, iterator last). Dazu müssen sie die Anfangs- und die Endposition des Containers als Parameter übergeben. Da begin() bzw. end() diese Positionen zurückliefern sind Sie bereits in der Lage den Inhalt des Container cont1 zu löschen!

☞ **Löschen Sie den Inhalt des Containers cont1!**
☞ **Testen Sie das Ergebnis des Löschvorgangs mit der Funktion size( )!**

☞ **Um weitere Funktionalitäten des Containers zu erfahren testen Sie die Funktionen void random_shuffle(…) sowie void replace(…) aus der Headerdatei <algorithm>!**