Hydroinformatik I - WiSe 2020/2021
HyBHW-S1-01-V8b: Container

Prof. Dr.-Ing. habil. Olaf Kolditz

[1]Helmholtz Centre for Environmental Research – UFZ, Leipzig
[2]Technische Universität Dresden – TUD, Dresden
[3]Center for Advanced Water Research – CAWR
[4]TUBAF-UFZ Center for Environmental Geosciences – C-EGS, Freiberg / Leipzig

Dresden, 11.12.2020

Semesterfahrplan

WiSe 2020/2021: Hydroinformatik I, Freitag (3. DS) 11:10-12:40, HÜL/S186/H

No	KW	Datum	ID	Vorlesung	Dozent
1	44	30.10.2020	HyBHW-1-01-01	Hydroinformatik - Einführung	Kolditz
2	44	30.10.2020	HyBHW-1-01-02	Compiler (Installation)	Kolditz
3	45	06.11.2020	HyBHW-1-01-03	Jupyter, Python	Kolditz
4	46	13.11.2020	HyBHW-1-01-04	Datentypen	Rink
5	47	20.11.2020	HyBHW-1-01-05	Klassen	Kolditz
6	48	27.11.2020	HyBHW-1-01-06	Input-Output (I/O)	Kolditz
7	49	04.12.2020	HyBHW-1-01-07	Strings - Textverarbeitung	Kolditz
8	50	11.12.2020	HyBHW-1-01-08	Pointer & Container	Kolditz
9	51	18.12.2020	HyBHW-1-01-09	Christmas Lecture	
10	1	08.01.2021	HyBHW-1-01-10	Hydrologische Modellierung	Kolditz
11	2	15.01.2021	HyBHW-1-01-11	BigData & Water 4.0	Kolditz
12	3	22.01.2021	HyBHW-1-01-12	Neuronale Netzwerke	Kolditz
13	4	29.01.2021	HyBHW-1-01-13	ANN / Bayes'sche Netzwerke	Kolditz
14	5	05.02.2021	HyBHW-1-01-14	BN / Maschinelles Lernen	Kolditz
15				Klausurvorbereitung	Kolditz

Informatik und Tools
Programmieren in C++
Hydrologische Modellierung

Fahrplan für heute ...

- bisher nur mit einzelnen Elementen (Zahlen, Text, Klassen) beschäftigt
- stimmt nicht ganz: Felder (Arrays) `double*` etc.
- heute: Arbeiten mit vielen Elementen

Fahrplan für heute ...

- bisher nur mit einzelnen Elementen (Zahlen, Text, Klassen) beschäftigt
- stimmt nicht ganz: Felder (Arrays) `double*` etc.
- heute: Arbeiten mit vielen Elementen

Fahrplan für heute ...

- bisher nur mit einzelnen Elementen (Zahlen, Text, Klassen) beschäftigt
- stimmt nicht ganz: Felder (Arrays) `double*` etc.
- heute: Arbeiten mit vielen Elementen

STL `std::`

Wir haben bereits mit der **String-Klasse** gesehen, dass C++ neben der Basisfunktionalität einer Programmiersprache auch sehr nützliche Erweiterungen, wie das Arbeiten mit Zeichenketten, anbietet. Diese Erweiterungen gehören zum Standard von C++, gehören also zum 'Lieferumfang' des Compilers dazu und heißen daher auch Standard-Klassen. Diese Klassen bilden die sogenannte **Standard-Bibliothek** (STL - STandard Library).

Container

In diesem Kapitel beschäftigen wir uns mit sogenannten **Containern** mit denen Daten organisiert, gespeichert und verwaltet werden können, z.B. Listen und Vektoren. In der Abbildung sind die wichtigsten Elemente der Container dargestellt. Der Begriff Container soll verdeutlichen, dass Objekte des gleichen Typs z.B. in einer Liste gespeichert werden. Und wie immer bei Klassen, werden natürlich auch die Methoden für den Zugriff oder die Bearbeitung von diesen Objekten bereitgestellt. Die Speicherverwaltung für Container-Elemente erfolgt dynamisch zur Laufzeit (siehe Kapitel Speicherverwaltung). Die Abbildung zeigt auch, welche verschiedenen **Typen** von Containern es gibt:

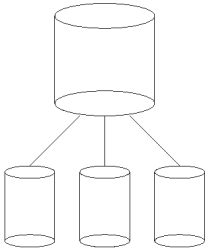
Container

(größtes Containerschiff, über 400m lang, 60 m breit, 19224 Container, ...)

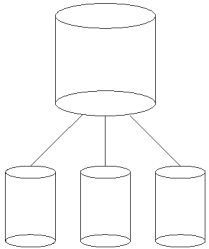
Container

Container

sequential Container



associative Container



Container

Sequentielle Container: sind z.B. Vektoren, Listen, Queues und Stacks. Vektoren sind Felder (arrays) in denen die einzelnen Elemente nacheinander angeordnet sind. Bei einer Liste kennt jedes Element nur seine Nachbarn (Vorgänger- und Nachfolgeelement). Queues (wir kennen dies von Druckern) sind Warteschlangen, die nach dem FIFO-Prinzip (first in, first out) funktionieren. Das heißt, dass zuerst eingefügte Element wird auch zuerst verarbeitet (z.B. Druckjobs). Stacks sind sogenannte Kellerstapel, die nach dem LIFO-Prinzip (last in, first out) funktionieren. Das bedeutet, das zuletzt eingefügte Element wird zuerst verarbeitet.

Assoziative Container: sind z.B. Maps (Karten) und Bitsets. Die Elemente sind nicht wie bei sequentiellen Container linear angeordnet sondern in bestimmtem Strukturen (z.B. Baumstrukturen). Dies ermöglicht einen besonders schnellen Datenzugriff, der über eine Verschlüsselung erfolgt.

Container

Die Besonderheit der C++ Container ist die **dynamische Speicherverwaltung**. Das heisst, Einfügen und Entfernen von Container-Elementen ist sehr einfach, wir brauchen uns nicht selbst um das Speichermanagement zu kümmern. Dies übernehmen die Konstruktoren und Destruktoren der jeweiligen Container-Klasse.

Sequentielle Container

Klassen-Template	Include-Files
vector<T,Allocator>	<vector>
list<T,Allocator>	<list>
stack<T,Container>	<stack>
queue<T,Container>	<queue>

Table: Klassentemplates für sequentielle Container

Container

Ein Klassentemplate ist nichts weiter als eine formale Beschreibung der Syntax.

- vector: Schlüsselwort
- $<$ ... $>$: Parameterliste
- T: erster Parameter, Datentyp des Vektors
- Allocator: zweiter Parameter, Speichermodell des Vektors

Container

Die Methoden sequentieller Container sind sehr einfach, intuitiv zu bedienen und sie sind gleich für Vektoren, Listen etc. Ein Auswahl der wichtigsten gemeinsamen Methoden finden sie in der nachfolgenden Tabelle.

Methode	Bedeutung
`size()`	Anzahl der Elemente, Länge des Containers
`push_back(T)`	Einfügen eines Elements am Ende
`pop_back()`	Löschen des letzten Elements
`insert(p,T)`	Einfügen eines Elements vor p
`erase(p)`	Löschen des p-Elements (an der Stelle p)
`clear()`	Löscht alle Elemente
`resize(n)`	Ändern der Containerlänge

Table: Methoden für sequentielle Container

Vektoren

The standard vector is a template defined in `namespace std` and presented in `<vector>`. The vector container is introduced in stages: member types, iterators, element access, constructors, list operations etc. The structure of a vector is illustrated in Fig. 1.

Figure: Structure of vectors

Vektoren

Following lines are necessary in order to use the vector container in your sources code.

```
1 #include <vector>
2 using namespace std;
```

Vektoren

Types: Very different types can be used in vectors. A vector e.g. of integers can be declared in this way.

```
1 // Declaration and construction
2   vector<int> v;
```

Vektoren

Stack operators: The functions `push_back` and `pop_back` treat the vector as a stack by adding and removing elements from its end.

```
1 // Input elements to vector
2   for(i=0;i<9;i++)
3   {
4     v.push_back(i+1);
5   }
```

Container

Iterators: can be used to navigate containers without the programmer having to know the actual type to identify the elements, e.g. for inserting and deleting elements. A few number of key functions allow to step through the elements, such as `begin()` pointing to first element and `end()` pointing to the one-past-last element. An example of a loop through all vector elements using the iterator is given below.

```
1 // Navigating vector elements
2   for(vector<int>::iterator it=v.begin(); it!=v.end(); ++it)
3   {
4     cout << *it << '\n';
5   }
```

Vektoren: Zugriff

Vector elements can be addressed directly by element number. Fast data access is an advantage of vectors.

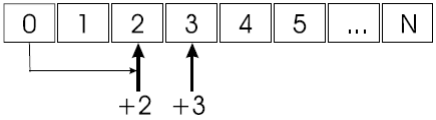


Figure: Access to vector elements

Vektoren: Elemente löschen

In contrast, vectors are not as flexible as lists. Deleting and adding of elements within the vector is complicated (Fig. 3).

Figure: Deleting elements

Vektoren: Übungen

- EX08b-vector: Vektoren anlegen
- EX08b-data-base: Verwalten von Datenbank-Objekten (Studenten-Klasse)

Vektoren: Übung EX08b

```
...
#include <vector>    // for using vectors
using namespace std; // for std functions

int main()
{...
  vector<long>my_vector;
  //-----------------------------------------------------------------
  cout << "Current vector size is: " << my_vector.size() << endl;
  for(long i=1;i<101;i++)
  {
     my_vector.push_back(i*i*i*i);
  }
  cout << "Current vector size is: " << my_vector.size() << endl;
  //-----------------------------------------------------------------
  for(long i=0;i<(long)my_vector.size();i++)
  {
    cout << my_vector[i] << endl;
  }
  //-----------------------------------------------------------------
  my_vector.clear();
  cout << "Current vector size is: " << my_vector.size() << endl;
...}
```

Vektoren

In der nächsten Übung werden einige Dinge, die wir bisher gelernt haben, zusammenbringen. Sie werden sehen, dass 'gut' programmierte Dinge einfach 'zusammengebaut' werden können: Objekte, IO-Methoden, Stringverarbeitung und natürlich Container.

Vektoren

```
#include <iostream>  // for using cout
#include <fstream>   // for using ifstream / ofstream
#include <string>    // for using string
#include <vector>    // for using vectors
#include "student.h" // for using CStudents
using namespace std; // for std functions
```

Listen

A list is a sequence optimized for insertion and deletion of elements. They allow a very flexible organization of elements. But the price for flexibility is a relatively slow access to data.
List provides bidirectional iterators. This implies that a STL list will typically be implemented using some form of a doubly-linked list.

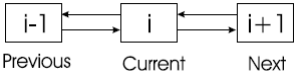


Figure: Structure of lists

Listen

Following lines are necessary in order to use the list container in your sources code.

```
1 #include <list>
2 using namespace std;
```

Arbeiten mit Listen

- Übung E83
- Studenten-Datenbank lesen: Übung EX08b-data-base/E84
- Warum: Erweiterung der Übung (Verknüpfen von Vektoren und Listen-Funktionalitäten)

Ende des Programmierteils (C++)