Kurseinheit 9: Templates

1. Motivation
2. Container-Klassen
3. Iteratoren
4. Algorithmen

1. Motivation

Datenstrukturen und Container und Collections

Bisher wurden **Datenstrukturen** wie Ringbuffer, Stack und ein eigenes Array implementiert bzw. im Rahmen der Lehrveranstaltung vorgestellt. Mittels Templateklassen konnten diese auch noch **generisch** sein. Erfüllen die eigenen Klassen wie Complex die Anforderungen der Templateklassen, so können diese auch als „Input“ verwendet werden.

(Template-)Klassen, denen verschiedene Elemente (Objekte, Zeiger, Referenzen) hinzugefügt und wieder „weggenommen“ werden können, werden als **Container** bezeichnet.

Mit dem Aufkommen der GUI wurden Container namentlich eher für grafische Container (Panels, …) verwendet, welche weitere grafische Elemente enthalten können. Für die nicht-grafischen Container wird daher eher der Begriff des **Collections** verwendet.
In der Programmiersprache Java werden Container im Umfeld der GUI-Programmierung verwendet, während Collections als Oberbegriff für die generischen Datenstrukturen dienen.
In C++ (ISO C++ Standard) scheint es diese scharfe Trennung nicht zu geben.

Oft finden sich in der C++-Literatur beide äquivalenten Begriffe.

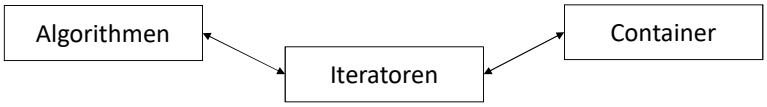
1. Motivation

Bibliotheken

Sinnvoll wäre eine Bibliothek, welche Templateklassen bereits enthält, die sofort genutzt werden können: Zeitersparnis und sicherlich sind weniger Fehler darin enthalten!

Microsoft entwickelte beispielsweise die MFC (Microsoft Foundation Classes). Es gab aber auch andere Ansätze.

So entwickelten Alexander Stepanov und Meng Lee mit ihren Kollegen bei Hewlett Packard in den achtziger Jahren eine Bibliothek: **Die Standard Template Library** (STL). Das Konzept kann durch eine Trennung von Container und Algorithmen überzeugen. Mittels Iteratoren wird auf die Container zugegriffen:

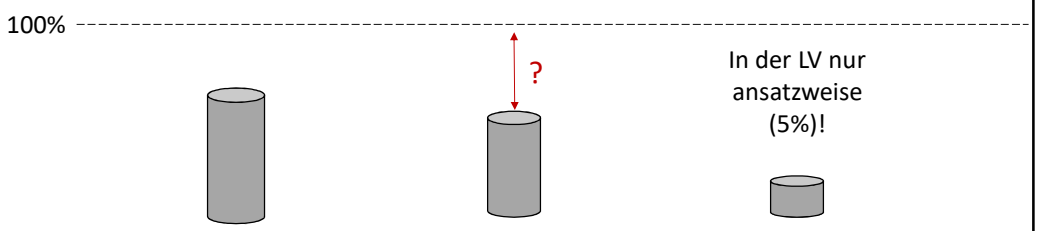


Die STL wurde 1998 in den C++-Standard aufgenommen und wird stetig weiterentwickelt.

1. Motivation

„Programmierlevel“

Wie gut (oder wie schnell) kann jemand programmieren?
Dazu sind die drei folgenden Säulen notwendig!



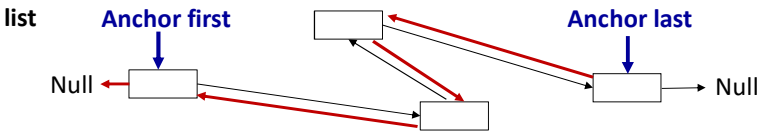
Kenntnisse Sprache C++	**Abstraktes und logisches Denken**	**Erfahrungen mit der STL**
Fakultät EMI: 40-70%	**Individuell** Man kann es oder nicht!	Lernt man im ersten Berufsjahr

Arten von Containern

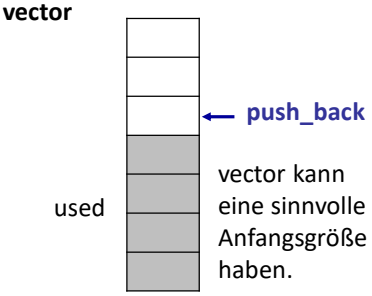
Container-Art	Header	Template	Sonstiges
Sequenzen Elemente liegen hintereinander	<array>	array	Nachbildung eines Arrays
	<deque>	deque	FIFO, Schreiben auch am Anfang
	<forward_list>	forward_list	Einfach verkettete Liste
	<list>	list	Doppelt verkettete Liste
	<queue>	queue	FIFO
	<stack>	stack	LIFO
	<vector>	vector	Bei Unwissenheit: Standard
Sortierte assoziierte Container	<map>	map, multimap	Key-Value: unique/not unique
	<set>	set, multiset	Value: unique/not unique
Ungeordnete assoziierte Container	<unordered_map>	unordered_map	
		unordered_multimap	
	<unordered_set>	unordered_set	
		unordered_multiset	
Spezialfälle	bitset, vector<bool>, priority_queue		

Sequenzen

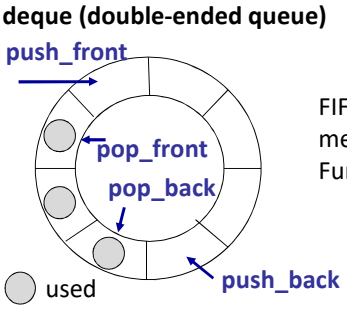
Die wichtigsten Sequenzen sind list, vector und deque.



Doppelt verkettete Liste



vector kann eine sinnvolle Anfangsgröße haben.



FIFO mit mehr Funktionalität

Komplexität der Sequenzen

Zugriffe auf die Sequenzen dauern in Abhängigkeit der Anzahl der Elemente n unterschiedlich lang. Ein erfahrener C++-Entwickler hat hier einen guten Überblick. In der Informatik wird diese Laufzeitkomplexität als O-Notation (Landau-Symbole) bezeichnet.
Beispiele:
O(n): Die Laufzeit verhält sich linear zur Anzahl der Elemente
O(1): Die Laufzeit ist unabhängig von der Anzahl der Elemente (ist konstant)

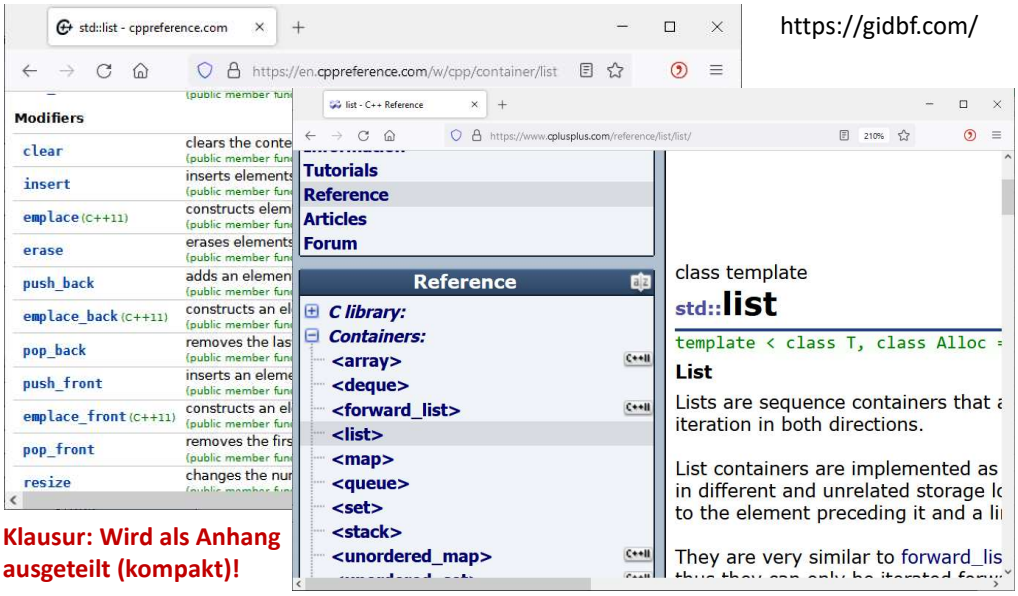
Details finden sich hier:

https://alyssaq.github.io/stl-complexities/

Oft ergeben sich aufgrund der internen Speicherung (Array oder Linked List) gerade für Anfänger unerwartete signifikante Unterschiede.

	Insert(iter, val)	Operator []
list	O(1)	O(n) Dieser ist aus Performanzgründen aber nicht implementiert.
vector	O(n)	O(1)

Sequenzen - Dokumentation



https://gidbf.com/

Klausur: Wird als Anhang ausgeteilt (kompakt)!

Beispiele Sequenzen: Vector mit int32_t

```
int32_t s32V1 = 5;
int32_t s32V2 = 6;
int32_t s32V3 = 7;
std::vector<int32_t> vectors32;
std::vector<int32_t*> vectorps32;

vectors32.push_back(s32V1);
vectors32.push_back(s32V2);
vectors32.push_back(s32V3);

vectorps32.push_back(&s32V1);
vectorps32.push_back(&s32V2);
vectorps32.push_back(&s32V3);
```

Container mit Referenzen als Parameter sind direkt nicht möglich.

https://stackoverflow.com/questions/33144419/stl-container-of-references-c11

Yes, the requirements are less strict now. However, right below the part you're referencing, the linked documentation *clearly states* that:

> The requirements that are imposed on the elements depend on the actual operations performed on the container. Generally, it is required that element type is a complete type and meets the requirements of Erasable, but many member functions impose stricter requirements. **(since C++11)**

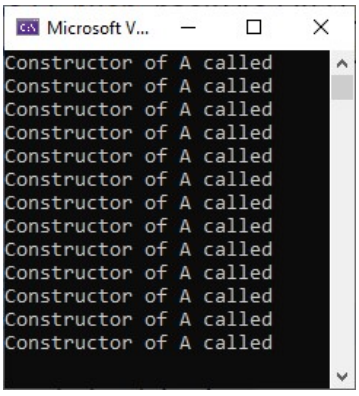
References don't meet the requirements of Erasable, therefore they still can't work.

Erasable:
Objekt kann über den Bezeichner gelöscht werden.
Mit Objekt und Pointer möglich, mit Referenz nicht.

Beispiele Sequenzen: Vector mit eigenem Objekt

```
A A1;
A A2;
A A3;

std::vector<A> vectorA(10);
std::vector<A*> vectorpA(10);

vectorA.push_back(A1);
vectorA.push_back(A2);
vectorA.push_back(A3);

vectorpA.push_back(&A1);
vectorpA.push_back(&A2);
vectorpA.push_back(&A3);
```

Hier werden beim Definieren eines Vectors schon 10 Elemente des Typs definiert (Speicher).
Es ist deutlich performanter, einmal eine größere Anzahl an Elementen zu allokieren, als ständig bei Bedarf immer ein Element hinzufügen.



In Summe werden hier 13 Objekte A instanziiert: A1, A2 und A3. In vectorA werden auch 10 Objekte A instanziiert. Beim Aufruf von vectorA.push_back werden dann nur die Inhalte kopiert.

Beispiele Sequenzen: Liste mit eigenem Objekt

```
A A1;
A A2;
A A3;

std::list<A> listA1(10);
std::list<A> listA2;
std::list<A*> listpa;

listA2.push_front(A1);
listA2.push_front(A2);
listA2.push_back(A2);
```

Hier werden beim Definieren einer Liste schon 10 Elemente des Typs definiert (Speicher) und miteinander verkettetet (Doppelt verkettete Liste). Eher unüblich.

Standardfall ist eher eine anfänglich leere Liste.

```
listpA.push_back(&A1);
listpA.push_back(&A2);
listpA.push_back(&A3);
listpA.push_back(&A1);
```

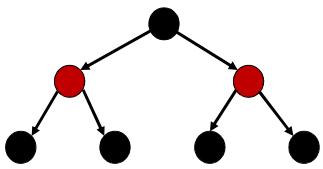
In einer Liste können aber auch Elemente mehrfach abgelegt werden. Gerade bei Pointern kann dies fatal sein: Über ein entnommenes Element wird Objekt zerstört (delete), der in der Liste verbleibende Pointer zeigt dann auf ungültigen Speicher.

Eine Liste hat nur dann einen Vorteil gegenüber einem Vector, wenn oft Elemente „zwischendrin“ eingefügt werden. Ansonsten ist der Vector einer Liste vorzuziehen.

Sortierte assoziierte Container

Die wichtigsten sortierten assoziierten Container sind map und set.

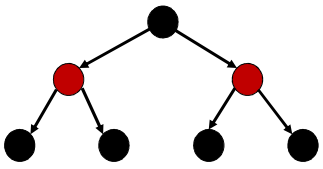
set



Rot-Schwarz-Baum:
Selbst balancierter Binärbaum.
Zugriff schnell:
$O(\log_2 n)$

Ein Objekt oder Zeiger kann nur einmal in einem set vorkommen (unique).

map



Rot-Schwarz-Baum:
Selbst balancierter Binärbaum.
Zugriff schnell:
$O(\log_2 n)$

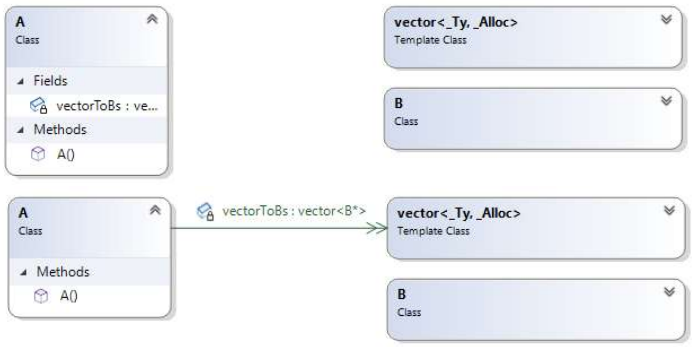
Bei einer map erhält noch das Element einen Wert (int32_t), nach dem wird dann sortiert (Key und Value).

Anwendung von Containern

Der Mercedes unter den Arrays ;-)

Container bieten umfangreiche Methoden, mit denen auf Array-like Strukturen zugegriffen werden kann . Ebenso können Container mit Algorithmen der STL genutzt werden (Sortieren, Reverse, Accumulate, Count mit Bedingungen, …).

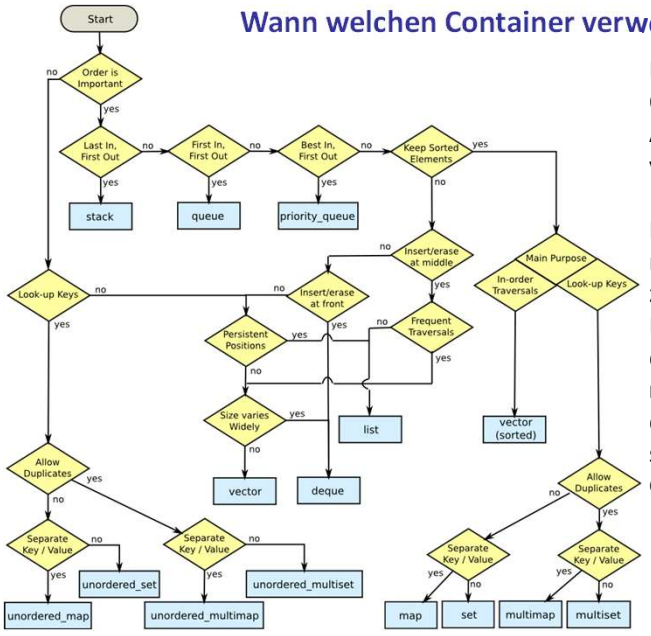
Abbildung von zu-n-Beziehungen (Elemente sind Pointer!)



vectorToBs:
Show As Field

vectorToBs:
Show As Collection Association

Wann welchen Container verwenden?



Beruht auf sehr viel Erfahrung! Ohne besondere Anforderungen ist meist der vector ausreichend.

Reichen die Elemente im vector nicht mehr aus, dann ist dieser zu vergrößern (**resize**()).
Eine Vergrößerung (**count**() gibt die aktuelle Anzahl zurück) mit **resize**() um viele Elemente ist deutlich schneller als ein ständiges Anwachsen der Größe mit **push_back**().

Zugriff auf Elemente eines Containers

Mit Iteratoren wird auf die einzelnen Elemente eines Containers zugegriffen. Viele Methoden eines Containers liefern einen passenden Iterator zurück, mit dem dann durch den Container „durch-iteriert“ wird. Jeder Container hat einen speziellen Datentyp für einen Iterator.

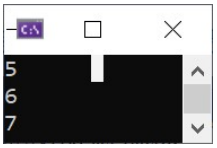
```
int32_t s32V1 = 5;
int32_t s32V2 = 6;
int32_t s32V3 = 7;

std::vector<int32_t> vectors32;

vectors32.push_back(s32V1);
vectors32.push_back(s32V2);
vectors32.push_back(s32V3);

for (std::vector<int>::iterator itvec = vectors32.begin();
     itvec != vectors32.end(); ++itvec)
{
   std::cout << *itvec << std::endl;
}
```

Definition passender Iterator, begin() gibt Iterator auf erstes Element im Container zurück.

Inkrementierung

end() gibt Iterator **nach** dem letzten Element (ähnlich NULL)

Dereferenzierung des Iterators



„Durch-Iterieren“ – leicht gemacht

Bisherige Methode:

```
for (std::vector<int>::iterator itvec = vectors32.begin();
     itvec != vectors32.end(); ++itvec)
{
   std::cout << *itvec << std::endl;
}
```

Einfachere Methode mit **auto ranged loop**:

```
for (auto ait : vectors32)
{
   std::cout << ait << std::endl;
}
```

Zukünftig immer diese Methode anwenden! Auch in der Klausur!

Deutlich einfacher:

- Datentyp inklusive Namespace entfällt.
- Inkrementierung entfällt.
- Der „merkwürdige“ Iterator, der hinter das letzte Element zeigt ist verschwunden (end()).
- Dereferenzierung des Iterators entfällt, ait hat schon den richtigen Typ des Elements.

Auto ranged loop auch mit Zeigern

```
int32_t s32V1 = 5;
int32_t s32V2 = 6;
int32_t s32V3 = 7;
```

```
std::vector<int32_t> vectors32;

vectors32.push_back(s32V1);
vectors32.push_back(s32V2);
vectors32.push_back(s32V3);
```

```
for (auto ait : vectors32)
{
   std::cout << ait << std::endl;
}
```

```
std::vector<int32_t*> vectorps32;

vectorps32.push_back(&s32V1);
vectorps32.push_back(&s32V2);
vectorps32.push_back(&s32V3);
```

```
for (auto ait : vectorps32)
{
   std::cout << *ait << std::endl;
}
```

Zu-n-Beziehungen werden mit Zeigern realisiert! Hier Dereferenzierungsoperator notwendig, um Wert auszugeben.

Konstante Iteratoren

Mit diesen kann nur lesend auf die Elemente zugegriffen werden.

```
for (std::vector<int>::iterator itvec = vectors32.begin();
     itvec != vectors32.end(); ++itvec)
{
   std::cout << *itvec << std::endl;
   *itvec += 1;
}

for (std::vector<int>::const_iterator itvec = vectors32.cbegin();
     itvec != vectors32.cend(); ++itvec)
{
   std::cout << *itvec << std::endl;
   //*itvec += 1; Fehler
}
```

Mit einem konstanten Iterator kann man Elemente in einem nicht konstanten Container nicht ändern.

Konstante Iteratoren

```
std::vector<int32_t> vectors32;
//..
test(vectors32);
```

Der übergebene Vector wird jetzt konstant!

```
void test(const std::vector<int32_t> cvectors32)
{
   for (std::vector<int>::iterator itvec = cvectors32.begin();
       itvec != vectors32.end(); ++itvec)
   {
      std::cout << *itvec << std::endl;
      *itvec += 1;
   }

   for (std::vector<int>::const_iterator itvec = cvectors32.cbegin();
       itvec != cvectors32.cend(); ++itvec)
   {
      std::cout << *itvec << std::endl;
   }
}
```

Compilerfehler!
Funktionen würden nicht konstanten Iterator zurückgeben.

Konstante Iteratoren: Häufiger Fehler

```
class B1
{
public:
   B1(const B1& rB)
   {
      for (std::vector<int>::iterator itvec = rB.vectors32_.begin();
          itvec != rB.vectors32_.end(); ++itvec)
      {
         vectors32_.push_back(*itvec);
      }
   }

private:
   std::vector<int32_t> vectors32_;
};
```

Compilerfehler: Return Value von rB.vector32_.begin () ist ein const_iterator! Kann nicht einem nicht konstanten Iterator zugewiesen werden. rB ist konstante Referenz!!!

https://www.cplusplus.com/reference/vector/vector/begin/

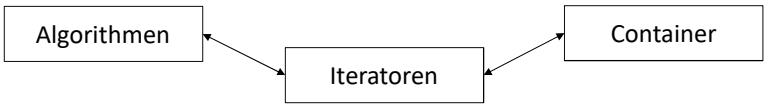
Return Value

An iterator to the beginning of the sequence container.

If the vector object is const-qualified, the function returns a `const_iterator`. Otherwise, it returns an `iterator`.

Prinzip in der STL

Die Algorithmen (Template-Algorithmen) der STL arbeiten nicht direkt auf den Containern (Abhängigkeiten), sondern nur über Iteratoren. Dadurch sind die Algorithmen universell einsetzbar.



Vereinzelt gibt es aber auch Algorithmen (z.B find), welche es in den speziellen Containern gibt. Es ist dann meist zu erwarten, dass diese deutlich performanter sind. Daher sollten dann im Zweifel diese „internen“ verwendet werden.

Algorithmen der STL lassen sich unterteilen in:

- **Nicht-verändernde Algorithmen:** Diese lassen die Elemente innerhalb eines Containers unverändert (Suche nach einem Element).
- **Verändernde Algorithmen:** Diese ändern die Elemente innerhalb des Containers ab (Sortieren eines Containers).

Beispiel find (Algorithmus)

```
#include <algorithm>
```

```
std::vector<int>::iterator itres = std::find(vectors32.begin(),
                                             vectors32.end(), 42);
if (itres != vectors32.end())
{
   std::cout << "found" << std::endl;
}
else
{
   std::cout << "not found" << std::endl;
}
```

Sobald das Element (hier 42) nicht gefunden wird, wird ein Iterator zurückgegeben, der hinter das letzte Element zeigt. Darauf kann abgefragt werden.

Komplexität dürfte **O(n)** sein.

vector hat selbst keine find-Methode.

Beispiel find als Methode in set

```
std::set<int32_t> sets32;
std::set<int32_t>::iterator itset;

sets32.insert(42);
sets32.insert(73);
sets32.insert(99);
itset = sets32.find(73);

if (itset != sets32.end())
{
   std::cout << "found" << std::endl;
}
```

find-Methode von set wird verwendet.

Selbstverständlich hätte hier auch mit auto ait gearbeitet werden könne!

Komplexität **O($\log_2$ n)**, also schneller als find (Algorithmen).

Könnte man mal nachmessen: Viele Elemente, einmal find von algorithm und von set. Messung ggf. mit feinerem Timer des Betriebssystems.