FH MÜNSTER
University of Applied Sciences

Programmieren in C++

Teil 8 – Die C++ Standardbibliothek

Prof. Dr. Kathrin Ungru
Fachbereich Elektrotechnik und Informatik

kathrin.ungru@fh-muenster.de

Die C++ Standardbibliothek

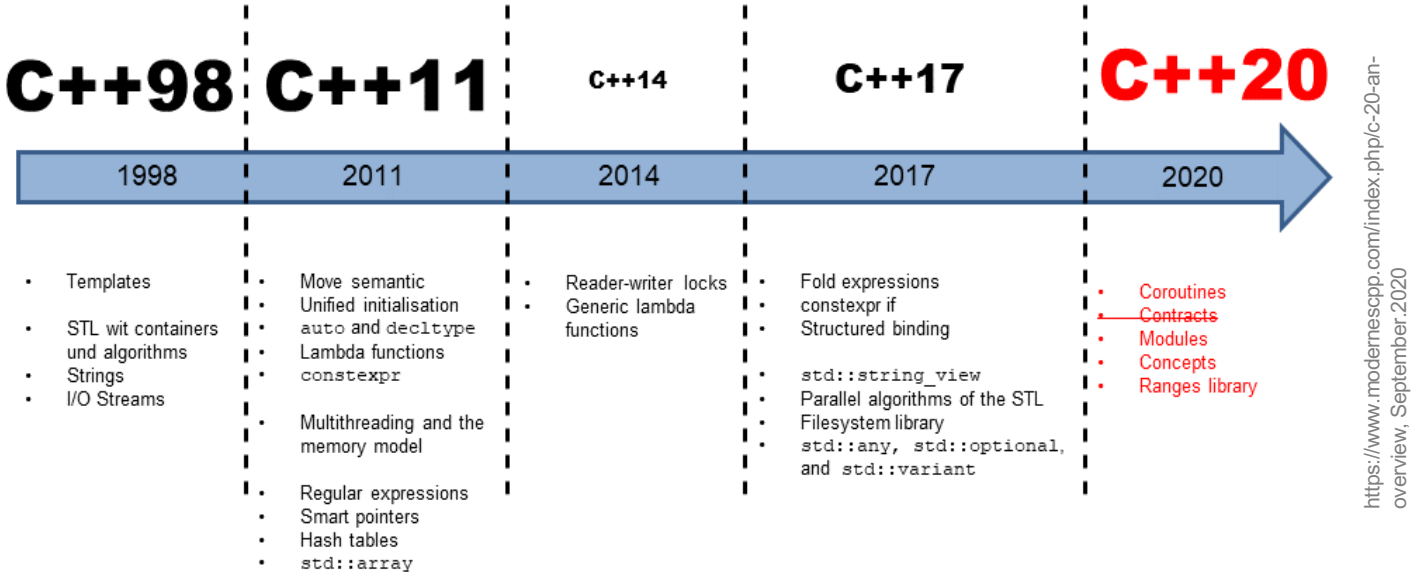
Inhalt

FH MÜNSTER
University of Applied Sciences

- Exkurs: Templates
- Exkurs: Das Schlüsselwort `auto`
- **Zeichenketten** mit der Strings-Bibliothek
- Ein- und Ausgabe mit **Streams**
- Datensammlungen verarbeiten mit
  - **Containern** und
  - **Iteratoren**

Die C++ Standardbibliothek

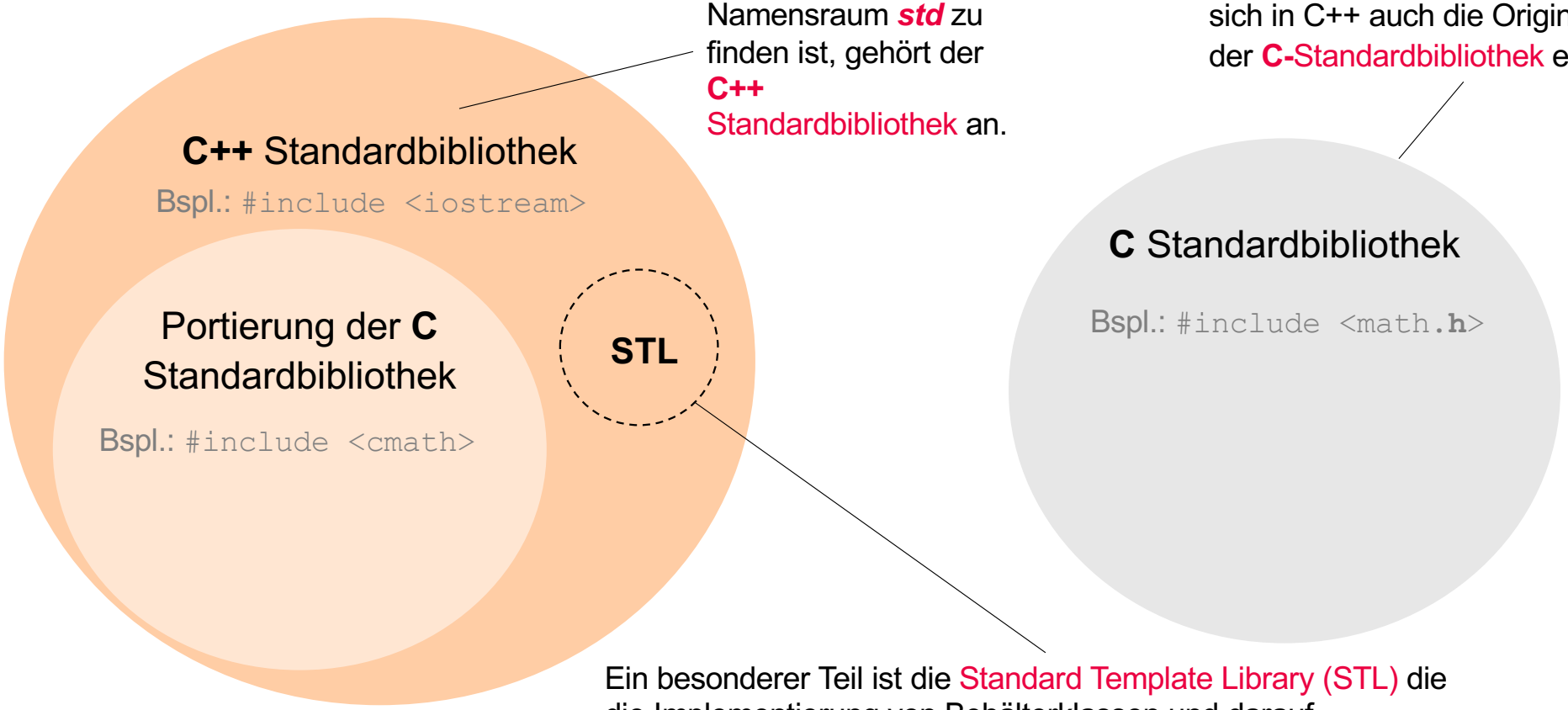
Der C++ Standard im Überblick



Der C++ Standard enthält neben Syntax und Schlüsselwörtern auch eine **Standardbibliothek**.

Die C++ Standardbibliothek

Überblick



Alles was im Namensraum ***std*** zu finden ist, gehört der **C++** Standardbibliothek an.

Aus Kompatibilitätsgründen lassen sich in C++ auch die Original-Header der **C**-Standardbibliothek einbinden.

Ein besonderer Teil ist die Standard Template Library (STL) die die Implementierung von Behälterklassen und darauf basierenden Algorithmen bereitstellt.

Die C++ Standardbibliothek

Umfang

C++ Standard Library headers

The interface of C++ standard library is defined by the following collection of headers.

Ausschnitt

Concepts library	
`<concepts>` (C++20)	Fundamental library concepts
Coroutines library	
`<coroutine>` (C++20)	Coroutine support library
Utilities library	
`<cstdlib>`	General purpose utilities: program control, dynamic memory allocation, random numbers, sort and search
`<csignal>`	Functions and macro constants for signal management
`<csetjmp>`	Macro (and function) that saves (and jumps) to an execution context
`<cstdarg>`	Handling of variable length argument lists
`<typeinfo>`	Runtime type information utilities
`<typeindex>` (C++11)	`std::type_index`
`<type_traits>` (C++11)	Compile-time type information
`<bitset>`	`std::bitset` class template
`<functional>`	Function objects, Function invocations, Bind operations and Reference wrappers
`<utility>`	Various utility components
`<ctime>`	C-style time/date utilites
`<chrono>` (C++11)	C++ time utilites
`<cstddef>`	Standard macros and typedefs
`<initializer_list>` (C++11)	`std::initializer_list` class template
`<tuple>` (C++11)	`std::tuple` class template
`<any>` (C++17)	`std::any` class
`<optional>` (C++17)	`std::optional` class template
`<variant>` (C++17)	`std::variant` class template
`<compare>` (C++20)	Three-way comparison operator support
`<version>` (C++20)	Supplies implementation-dependent library information
`<source_location>` (C++20)	Supplies means to obtain source code location

https://en.cppreference.com/w/cpp/header

Exkurs: Das Schlüsselwort `auto`

Schlüsselwörter in diesem Kapitel

alignas	const	dynamic_cast	long	short	typedef
alignof	consteval	else	mutable	signed	typeid
asm	constexpr	enum	namespace	sizeof	typename
auto	constinit	explicit	new	static	union
bool	const_cast	export	noexcept	static_assert	unsigned
break	continue	extern	nullptr	static_cast	using
case	co_await	false	operator	struct	virtual
catch	co_return	float	private	switch	void
char	co_yield	for	protected	template	volatile
char8_t	decltype	friend	public	this	wchar_t
char16_t	default	goto	register	thread_local	while
char32_t	delete	if	reinterpret_cast	throw	
class	do	inline	requires	true	
concept	double	int	return	try	

Exkurs: Das Schlüsselwort `auto`

Automatische Typerkennung


```
auto x{1}; // Typ eines Literals

Kreis kreis{3};
auto &rKreis = kreis; // Referenz

auto *pKreis1 = &kreis; //A: wird als "Kreis *pKreis1" erkannt
auto pKreis2 = &kreis; //B: wird ebenfalls als "Kreis *pKreis2" erkannt

auto foo() // als Rückgabetyp
{
   return 1.0;
}
```

ab **C++20** auch als Übergabetyp in Funktionen möglich

Exkurs: Das Schlüsselwort `auto`

Automatische Typerkennung

- **Es wird empfohlen, das Schlüsselwort `auto` zu verwenden!**
- **Warum?**
  - ***Stabilität***: Wird ein Datentyp an einer Stelle geändert wird die Änderung überall automatisch übernommen.
  - ***Effizienz***: Es erfolgt keine (aufwändige) implizite Typumwandlung.
  - ***Benutzerfreundlichkeit***: einfachere Handhabung und eine einfache Möglichkeit zur Definition von komplizierten Typen.
- **Nicht verwenden**, wenn nur ein ganz bestimmter Typ benötigt wird.

Exkurs: Das Schlüsselwort `auto`

in for-Schleifen

FH MÜNSTER
University of Applied Sciences

```
string str("Hallo");
```

Elemente nutzen (per Kopie):

```
for(auto c: str) {
  cout << c << endl;
}

//Ausgabe:
// H
// a
// l
// l
// o
```

Elemente nutzen (per Referenz):

```
for(const auto &c: str) {
  cout << c << endl;
}

//Ausgabe:
// H
// a
// l
// l
// o
```

Elemente verändern:

```
for(auto &c: str) {
  c = 'T';
}

cout << str << endl;
//Ausgabe:
// TTTTT
```

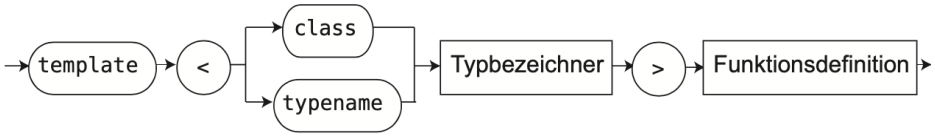
Exkurs: Templates

Schlüsselwörter in diesem Kapitel

alignas	const	dynamic_cast	long	short	typedef
alignof	consteval	else	mutable	signed	typeid
asm	constexpr	enum	namespace	sizeof	**typename**
auto	constinit	explicit	new	static	union
bool	const_cast	export	noexcept	static_assert	unsigned
break	continue	extern	nullptr	static_cast	using
case	co_await	false	operator	struct	virtual
catch	co_return	float	private	switch	void
char	co_yield	for	protected	**template**	volatile
char8_t	decltype	friend	public	this	wchar_t
char16_t	default	goto	register	thread_local	while
char32_t	delete	if	reinterpret_cast	throw	
class	do	inline	requires	true	
concept	double	int	return	try	

Exkurs: Templates

Template Funktionen



- Die folgende Template Funktion kann mit allen **Datentypen (Grundtypen, Klassen)** arbeiten:

`template` Schlüsselwort wird einer Funktion vorangestellt.

Funktions-implementierung

```
template <typename T>
void foo(T variable)
{
  ...
}
```

`<typename T>` definiert den Platzhalter `T`, der hier für einen **Datentypen** steht.
Stattdessen kann auch ein Platzhalter für einen **Wert** definiert werden, z.B. `<int N>` mit Platzhalter `N`, der hier für einen beliebigen konstanten Wert vom Typ `int` steht.

Funktionsaufruf

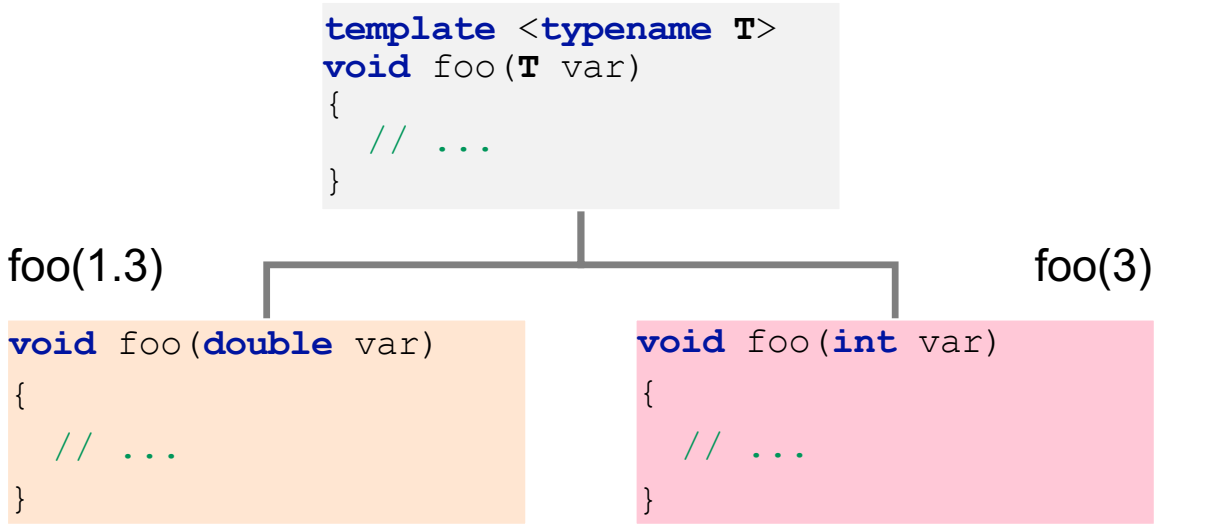
```
foo<double>(3.0);
foo(3.0); // Funktioniert auch
```

Exkurs: Templates

Template-Metaprogrammierung

- **Templates (**englisch: Vorlage, Schablone) erlauben es zur Kompilierzeit Quelltext so zu optimieren, dass Algorithmen speziell für bestimmte Datentypen ausgewählt werden

  - ➢ Die C++ Standardbibliothek besteht zu großen Teilen aus Templates.



Compiler generiert Funktion mit `double` Argument, wenn Double Zahl übergeben wird.

Compiler generiert Funktion mit `int` Argument, wenn Integer Zahl übergeben wird.

Exkurs: Templates

Template Klassen

Templates bieten eine Möglichkeit Klassen leicht auf andere Datentypen zu erweitern:

Deklaration

```
template <typename T>
class Vektor2D
{
  // Quelltext, z.B. 2D Vektor-Operationen

  private:
    T x;
    T y;
};
```

Instanziierung

```
Vektor2D<double> vektor;
```

Exkurs: Templates

Template Klassen

Eine Template Klasse für *n*-dimensionale Vektoren:

Deklaration

```
template <typename T, unsigned int N=2> // zwei Parameter
class Vektor
{
  // Quelltext, z.B. n-D Vektor-Operationen

  private:
    T elemente[N];
};
```

Instanziierung

```
Vektor<double>    vektor2D;
Vektor<double, 3> vektor3D;
```

Die C++ Standardbibliothek

Zeichenketten

Zeichenketten

Zeichenketten mit Strings

- Einbindung mit **`#include <string>`**
- Der Typ **`std::string`** ist definiert als **`std::basic_string`**<**`char`**>, d.h. eine Kette von einfachen 8-Bit Character-Zeichen.
- Vorteil der Template Klasse `basic_string`:
  - einfachere Handhabung von Zeichenketten.

Beispiel:

```
string str("Hallo");
str += " Welt";
cout << str << std;
// Ausgabe: Hallo Welt
```

- https://en.cppreference.com/w/cpp/string/basic_string

Zeichenketten

abarbeiten mit Bereichsbasierter for-Schleife


```
string str("Hallo");

for(char c: str) {
  cout << c << endl;
}

//Ausgabe:
// H
// a
// l
// l
// o
```

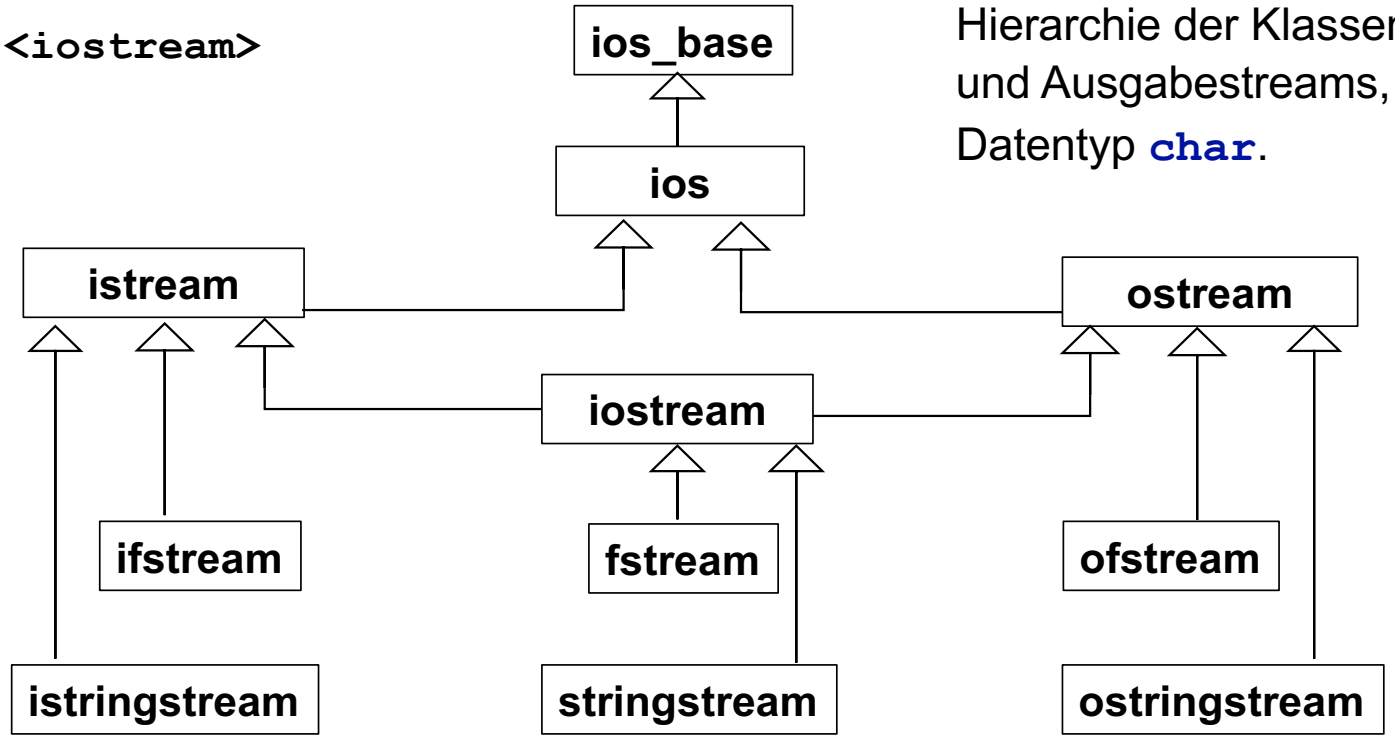
Die C++ Standardbibliothek

Ein- und Ausgabe-Streams

Ein- und Ausgabe-Streams

Übersicht

```
#include <iostream>
```



Hierarchie der Klassentemplates von Ein- und Ausgabestreams, hier speziell für den Datentyp `char`.

Ein- und Ausgabe-Streams

Ausgabestream: ostream

- Die Klasse `ostream` umfasst überladene Operatoren für einfache Datentypen und Datentypen der Standardbibliothek z.B. String, die mit Hilfe des Operators << in eine ASCII-Zeichenfolge verwandelt werden.

```
ostream& operator<<(const char*); // C-Strings
ostream& operator<<(int);
ostream& operator<<(double);
ostream& operator<<(std::string);
// usw.
```

- Der Operator << kann überladen werden, um eine Ausgabe von benutzerdefinierten Klassen zu ermöglichen.
- `cout`, `cerr` oder `clog` sind spezielle **Objekte** der Klasse `ostream`, die für bestimmte Ausgabe Aufgaben zur Verfügung stehen.

Ein- und Ausgabe-Streams

Eingabestream: istream

- Die Klasse `istream` sorgt mit Hilfe des Operators >> für eine Umwandlung der eingelesenen Zeichen in den richtigen Datentyp.

```
istream& operator>>(char*); // C-Strings
istream& operator>>(int&);
istream& operator>>(double&);
istream& operator>>(std::string&);
// usw.
```

- Der Operator >> kann überladen werden, um eine Eingabe von benutzerdefinierten Klassen zu ermöglichen.
- `cin` ist ein spezielles **Objekte** der Klasse `istream`, die für die Eingabe von Werten in der Konsole zur Verfügung steht.

Ein- und Ausgabe-Streams

Formatierung

FH MÜNSTER
University of Applied Sciences

Auszug

Name	Bedeutung
`boolalpha`	`true/false` statt 1/0 ausgeben
`left`	linksbündige Ausgabe
`right`	rechtsbündige Ausgabe
`dec`	dezimal
`oct`	oktal
`hex`	hexadezimal
`uppercase`	E,X statt e,x
`showpos`	+ bei positiven Zahlen anzeigen
`scientific`	Exponential-Format
`fixed`	Gleitkomma-Format
`endl`	neue Zeile ausgeben
`ws`	Zwischenraumzeichen endfernen

Beispiel:
```
cout << boolalpha << true;
```

Weitere Formatierung über Methoden von **ios** (Oberklasse von istream und ostream) möglich.

Container
Standard Template Library (STL)

- Entwickelt bei Hewlett-Packard von Alexander Stepanov, Meng Lee und Kollegen

- Wegen ihrer überzeugenden Konzeption von Behälterklassen sogenannte Container und Algorithmen von ISO-Komitee als Teil des C++ Standards aufgenommen.

- Die STL legt den Schwerpunkt auf generische Programmierung mit Templates und weniger auf Objektorientierung und Polymorphie.

- Warum Templates?
  - Vorteil: Auswertung von Templates erfolgt zu Kompilierzeit: Keine Laufzeiteinbußen durch dynamisch polymorphe Aufrufe.

Container
Beispiel: Vektor

```
#include <vector>
#include <iostream>

int main()  {
  std::vector<int> einContainer(100);
  // Container mit Werten füllen
  for(int i=0; i<einContainer.size(); ++i)
  {
    einContainer[i] = 2*i;
  }
  // Werte ausgeben
  for(int x: einContainer)
  {
    std::cout << x << " ";
  }
  std::cout << "\n";
  return 0;
}
```

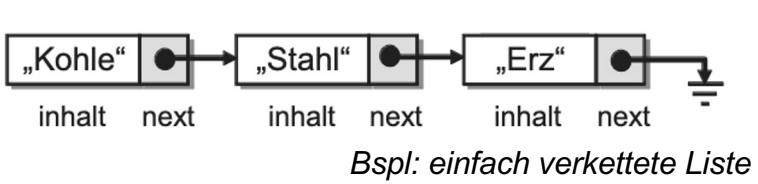
Container
Rückblick

FH MÜNSTER
University of Applied Sciences

- **Feld (Array):** Feste Größe

Bspl. Array

- **Liste (List):** Wächst dynamisch
  - Variante 1: Nicht verkettet
  - Variante 2: Einfach verkettet
  - Variante 3: Doppelt verkettet
  - Elementzugriff: Unterschiedliche Implementierungen

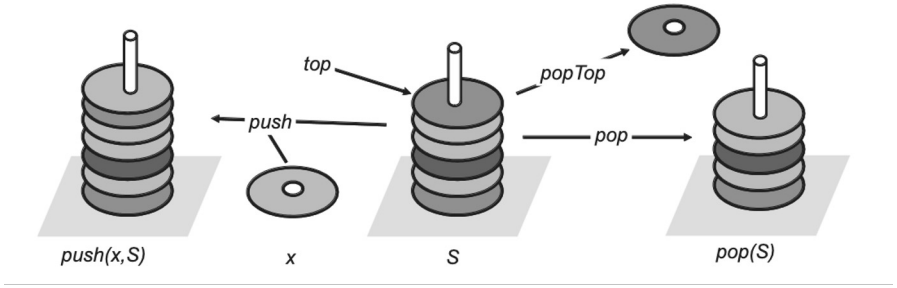


Bspl: einfach verkettete Liste

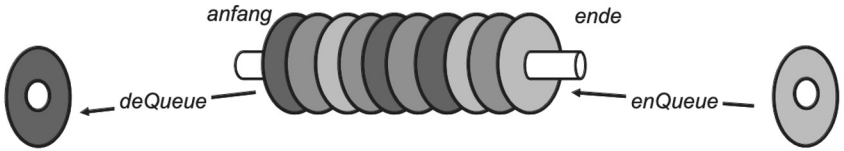
Quelle: Gumm, Sommer: Einführung in die Informatik. 10. Auflage, Oldenbourg 2012

Container
Rückblick

- **Stapel (Stack):** LIFO-Prinzip
  - push: lege Element auf Stapel
  - pop: hole Element von Stapel
  - Elementzugriff: top



- **Warteschlange (Queue):** FIFO-Prinzip
  - push (enQueue): hänge Element ans Ende
  - pop (deQueue): hole erstes Element
  - Elementzugriff: front, back



Quelle: Gumm, Sommer: Einführung in die Informatik. 10. Auflage, Oldenbourg 2012

Container
Überblick

Sequence containers

Sequence containers implement data structures which can be accessed sequentially.

array (C++11)	fixed-sized inplace contiguous array (class template)
vector	resizable contiguous array (class template)
inplace_vector (C++26)	resizable, fixed capacity, inplace contiguous array (class template)
hive (C++26)	collection that reuses erased elements' memory (class template)
deque	double-ended queue (class template)
forward_list (C++11)	singly-linked list (class template)
list	doubly-linked list (class template)

Associative containers

Associative containers implement sorted data structures that can be quickly searched ($O(\log n)$ complexity).

set	collection of unique keys, sorted by keys (class template)
map	collection of key-value pairs, sorted by keys, keys are unique (class template)
multiset	collection of keys, sorted by keys (class template)
multimap	collection of key-value pairs, sorted by keys (class template)

https://en.cppreference.com/w/cpp/container

Container
Überblick

Unordered associative containers (since C++11)

Unordered associative containers implement unsorted (hashed) data structures that can be quickly searched ($O(1)$ average, $O(n)$ worst-case complexity).

unordered_set (C++11)	collection of unique keys, hashed by keys (class template)
unordered_map (C++11)	collection of key-value pairs, hashed by keys, keys are unique (class template)
unordered_multiset (C++11)	collection of keys, hashed by keys (class template)
unordered_multimap (C++11)	collection of key-value pairs, hashed by keys (class template)

Container adaptors

Container adaptors provide a different interface for sequential containers.

stack	adapts a container to provide stack (LIFO data structure) (class template)
queue	adapts a container to provide queue (FIFO data structure) (class template)
priority_queue	adapts a container to provide priority queue (class template)
flat_set (C++23)	adapts a container to provide a collection of unique keys, sorted by keys (class template)
flat_map (C++23)	adapts two containers to provide a collection of key-value pairs, sorted by unique keys (class template)
flat_multiset (C++23)	adapts a container to provide a collection of keys, sorted by keys (class template)
flat_multimap (C++23)	adapts two containers to provide a collection of key-value pairs, sorted by keys (class template)

Views (since C++20)

Views provide flexible facilities for interacting with one- or multi-dimensional views over a non-owning array of elements.

span (C++20)	a non-owning view over a contiguous sequence of objects (class template)
mdspan (C++23)	a multi-dimensional non-owning array view (class template)

https://en.cppreference.com/w/cpp/container

Iteratoren

Für den Zugriff auf Container-Elemente

- Flexiblere Möglichkeit durch Datenstrukturen zu "wandern".
- **Idee:** Es soll der Reihe nach auf Elemente der Datenstruktur zugegriffen werden können, ohne die innere Struktur des Containers zu kennen.
- Arbeitet wie ein Zeiger mit erweiterter Funktionalität.

Beispiel:
Deklaration eines Iterators des **`list`**-Containers

```
std::list<double>::iterator itr;
```

Iteratoren

Beispiel: Iteratoren und Algorithmen

```
#include <algorithm> // algorithm einbinden für find()
//...
```

```
// ... Vektor initialisieren wie in vorherigem Beispiel
// ...

std::vector<int>::iterator begin = einContainer.begin();
std::vector<int>::iterator end = einContainer.end();
std::vector<int>::iterator pos = std::find(begin, end, 6); // Zahl suchen

cout << *pos << endl; // Inhalt an der Position des Iterators pos
cout << pos-begin << endl; // Index der gesuchten Zahl
cout << einContainer[pos-begin] << endl; //erhalte Vektorelement über Index
// Ausgabe:
// 6
// 3
// 6
```

Iteratoren

Beispiel: einfacher mit **auto**

```
#include <algorithm> // algorithm einbinden für find()
//...
```

```
// std::vector<int>::iterator wird durch auto ersetzt
// dadurch lässt sich die Art des Containers nach Bedarf ändern ohne
// stark den Quelltext verändern zu müssen
auto begin = einContainer.begin();
auto end = einContainer.end();
auto pos = std::find(begin, end, 6); // find() ist auf alle Container anwendbar

cout << *pos << endl; // Inhalt an der Position des Iterators pos
cout << pos-begin << endl; // Index der gesuchten Zahl
cout << einContainer[pos-begin] << endl; //erhalte Vektorelement über Index
// Ausgabe:
// 6
// 3
// 6
```

FH MÜNSTER
University of Applied Sciences

Programmieren in C++

Prof. Dr. Kathrin Ungru
Fachbereich Elektrotechnik und Informatik

kathrin.ungru@fh-muenster.de