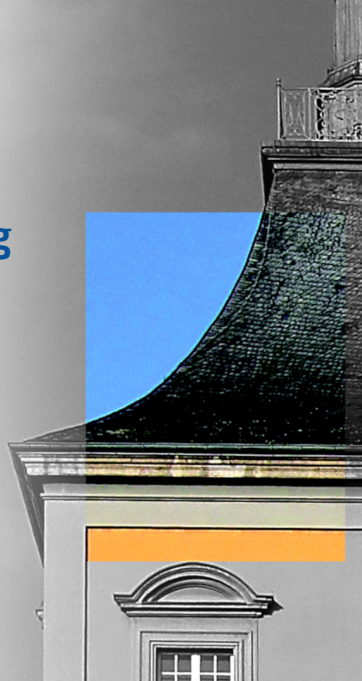
UNIVERSITÄT BONN

Algorithmen und Programmierung

Algorithmen II

Dr. Felix Jonathan Boes
boes@cs.uni-bonn.de
Institut für Informatik



KURZE WIEDERHOLUNG

Abstrakte Datentypen und Datenstrukturen

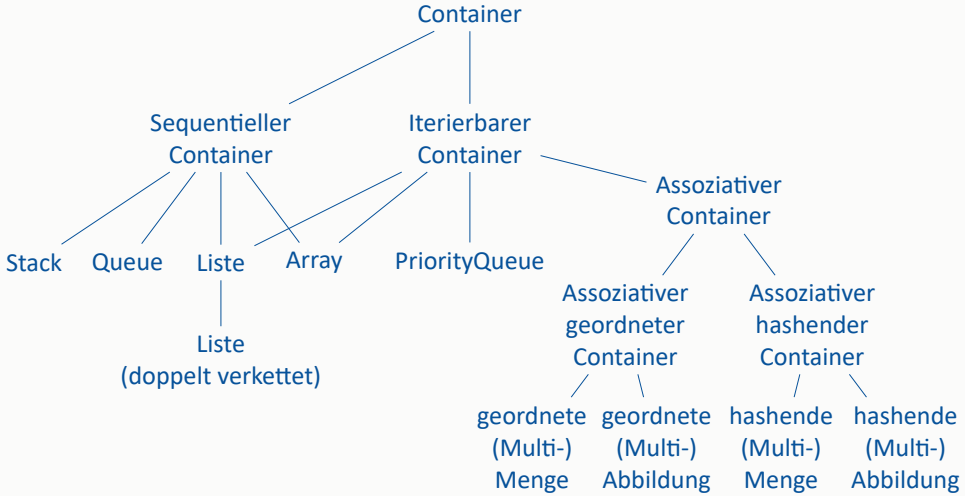
Offene Fragen

Wie sind die wichtigsten abstrakten Datentypen definiert?

Durch welche Datenstrukturen werden sie realisiert?

UNIVERSITÄT BONN

Die (wichtigsten) abstrakten Datentypen im Überblick



UNIVERSITÄT BONN

Assoziative Container

Geordnete (Multi)mengen organisieren untereinander vergleichbare Keys und werden von **AVL-Bäumen** realisiert.

Hashende (Multi)mengen organisieren hashbare Keys und werden von **Hashtabellen** realisiert.

(Multi)abbildungen speichern Key-Value-Paare und werden, entlang der zugehörigen Keys, analog zu (Multi)mengen realisiert.

Abstrakte Datentypen und Datenstrukturen

Wo und wie gibt man eine totale Ordnung oder eine Hashfunktion an?

Offene Frage

Wo und wie gibt man eine totale Ordnung oder eine Hashfunktion an?

UNIVERSITÄT BONN

Wo gibt man totale Ordnungen oder Hashfunktionen an?

Um assoziative Container zu definieren, muss eine totale Ordnung bzw. eine Hashfunktion angegeben werden.

Die offensichtliche Frage ist, ob dieses Datum (totale Ordnung bzw. Hashfunktion) eine **intrinsiche Eigenschaft** der Objekte ist oder ob es eine **zusätliche Information** ist, die man dem Container mitteilt.

UNIVERSITÄT BONN

Angabe in typischen objektorientierten Sprachen

In den Containern der Standardbibliothek von Java kann die totale Ordnung Teil des Objekts oder des Containers sein. Die Hashfunktion ist immer Teil des Objekts und nicht nutzerdefinierbar.

In den Containern der Standardbibliothek von Python gibt es keine geordneten Container. Die Hashfunktion ist immer Teil des Objekts und nutzerdefinierbar.

In den Containern der Standardbibliothek von C++ kann die totale Ordnung Teil des Objekts[1] oder des Containers sein. Die Hashfunktion ist immer Teil des Containers und nutzerdefinierbar.

[1] Dies geschieht implizit durch die Implementierung der Memberfunktion: `bool TYP::operator< (const TYP &other) const`

UNIVERSITÄT BONN

Funktionen als Parameter?

Eine totale Ordnung $\preceq$ ist so aussagekräftig wie eine zweistellige `compare`-Funktion. Dabei legen wir Folgendes fest.

$$X \prec Y \Leftrightarrow \texttt{compare(X,Y)} < 0$$
$$X \succ Y \Leftrightarrow \texttt{compare(X,Y)} > 0$$
$$\text{sonst} \Leftrightarrow \texttt{compare(X,Y)} = 0$$

Mit dieser Modellierung hängen Sortierverfahren und geordnete Multimengen von einer `compare`-Funktion als Parameter ab. Wir wünschen uns somit Sortierfunktionen zu schreiben, welche die folgende Signatur haben.

```
void sortieren(std::vector<TYP>&, compare-funktion)
```

Analog wollen wir assoziativen Containern beim Erzeugen eine Vergleichs- oder Hash-funktionen übergeben.

Haben Sie Fragen?

Zwischenfazit

Man übergibt totale Ordnungen oder Hashfunktionen als Funktionen

Wir lernen nun, wie man Funktionen als Parameter übergibt

Exkurs: Funktionsobjekte

Offene Frage

In wie weit sind Funktionen auch Objekte?

Wie kann man in C++ Funktionen als Parameter übergeben?

UNIVERSITÄT BONN

Funktionsobjekt

Einfach gesagt ist ein Objekt ein **Funktionsobjekt**, falls es wie eine Funktion verwendet werden kann. Genauer gesagt ist ein Objekt ein **Funktionsobjekt**, falls das Objekt den **Funktionsaufrufoperator** implementiert.

In objektorientierten Sprachen verwendet man Funktionsobjekte um Funktionen (und alles was sich so verhält) als Parameter an andere Funktionen zu übergeben. Beispiele sind Hashfunktionen oder die oben genannten `compare`-Funktionen.

UNIVERSITÄT BONN

Funktionsobjekte in objektorientierten Sprachen

In Java sind Objekte Funktionsobjekte, falls sie genau eine Funktion besitzen und das `Callable`-Interface implementieren.

In Python sind Objekte Funktionsobjekte, falls sie die folgende Memberfunktion implementieren.

```
__call__(self, /* Parameter */ )
```

In C++ sind Objekte Funktionsobjekte, falls sie die folgende Memberfunktion implementieren.

```
RUECKGABETYP TYP::operator()( /* Parameter */ )
```

In allen Fällen kann ein Funktionsobjekt `fobj` anschließend wie folgt verwendet werden.

```
fobj( /* Parameter */ )
```

UNIVERSITÄT BONN

Beispiel

```
// HEADERDATEI
class Zaehler {
public:
  Zaehler(int start); // Konstruktur
  int operator() (void);  // Gibt zurück wie oft der Zaehler verwendet wurde

private:
  int anzahl;
};

// QUELLDATEI
Zaehler::Zaehler(int start) : anzahl(start) {}
int Zaehler::operator() (void) { return anzahl++; }

// DEMO
int main() {
  Zaehler v(100); // Zaehler ist ein Funktionsobjekt
  std::cout << v() << " " << v() << " " << v() << std::endl;  // Druckt: '100 101 102'
}
```

UNIVERSITÄT BONN

Funktionen als Funktionsobjekt

Einfach gesagt kann in C++ jede Funktion als Funktionsobjekt aufgefasst werden. Zu einer Funktion, welche die Parameter `PAR1, PAR2, ...` entgegen nimmt und eine Rückgabe vom Typ `RUECK` produziert , hat das zugehörige Funktionsobjekt den folgenden Typ.

`std::function<RUECK(PAR1, PAR2, ...)>`

```
#include <functional> // Wird eingebunden um mit std::function zu arbeiten

int  f() {...}
bool g(int x, int y) {...}

// Erzeugt ein (leeres) Funktionsobjekt das keine Parameter erhält und int-Werte produziert
std::function<int(void)> fobj1
fobj1 = f; // fobj1 speichert ein Funktionsobjekt dass zur Funktion f gehört
// Erzeugt ein (leeres) Funktionsobjekt das zwei int-Parameter erhält und bool-Werte produziert
std::function<bool(int, int)> fobj2;
fobj2 = g; // fobj2 speichert ein Funktionsobjekt dass zur Funktion g gehört
```

UNIVERSITÄT BONN

Beispiel I
Zufallszahlengenerator als Parameter

Aus gewissen Gründen brauchen wir eine Funktion, die einen Array mit zufälligen Zahlen füllt und als letztes Element die Summe der Elemente schreibt. Dabei soll die Zufallsfunktion austauschbar sein.

```
void befuelle (std::vector<int>& zahlen, std::function<int(void)> zufall) {
  int summe = 0;
  for (int i = 0; i < zahlen.size() - 1; ++i) {
    zahlen[i] = zufall();
    summe += zahlen[i];
  }
  if (zahlen.size() > 0) { zahlen[zahlen.size() - 1] = summe; }
}

int main() {
  std::vector<int> coole_zahlen(10);
  befuelle(coole_zahlen, meine_random_fkt); // Befuelle mit meiner Random-Funktion
  fmt::print("Zahlen: {}\n", coole_zahlen);
  befuelle(coole_zahlen, deine_random_fkt); // Befuelle mit deiner Random-Funktion
  fmt::print("Zahlen: {}\n", coole_zahlen);
}
```

UNIVERSITÄT BONN

Beispiel II
Vergleichsfunktionen für sort

Ein wiederkehrendes Problem ist das Sortieren von Objekten. Die meisten Sortieralgorithmen brauchen dazu nur eine Vergleichsfunktion (die nur vom Objekttyp abhängt). Die Standardbibliothek gibt uns die sehr effiziente Sortierfunktion `std::sort`.

Die Funktion `std::sort` ist (vereinfacht gesagt) so wie hier definiert.

```
#include <algorithm> // Wird eingebunden um sort verwenden zu können
void sort (
  Iterator first, // Iterator der auf das erste Element zeigt
  Iterator last,  // Iterator der hinter das letzte Element zeigt
  // Vergleichsfunktion die zu true auswertet, genau dann wenn der erste Param kleiner ist
  std::function<bool(const T&, const T&)> comp,
);
```

UNIVERSITÄT BONN

Beispiel II
Vergleichsfunktionen für sort

Um Strings nach Ihrer Länge zu verlgeichen, implementieren wir folgende Vergleichsfunktion:

```
// Wertet zu true aus, genau dann wenn der erste String kürzer ist
bool str_comp(const std::string& a, const std::string& b) {
  if (a.size() < b.size()) {
    return true;
  } else {
    return false;
  }
}

int main() {
  std::vector<std::string> coole_strings {"Hi", "Ada", "Lovelace", "ist", "cool"};
  fmt::print("Strings: {}\n", coole_strings);
  std::sort(coole_strings.begin(), coole_strings.end(), str_comp);
  fmt::print("Strings: {}\n", coole_strings);
}
```

UNIVERSITÄT BONN

Funktionsobjekte und std::function

Bereits bestehende Funktionsobjekte können auch als Instanz von `std::function` aufgefasst werden. Zu einem Funktionsobjekt, welches Parameter `PAR1, PAR2, ...` entgegen nimmt und eine Rückgabe vom Typ `RUECK` produziert, hat die zugehörige Instanz von `std::function`, vereinfacht gesagt, den folgenden Typ.

`std::function<RUECK(PAR1, PAR2, ...)>`

```
#include <functional> // Wird eingebunden um sort verwenden zu können

class MeinFunktionsobjekt {
public:
  // Produziert einen String mit anz vielen Worten
  std::string operator() (int anz);
};
...
  MeinFunktionsobjekt mein_fobj;                    // Typ: MeinFunktionsobjekt
  std::function<std::string(int)> fobj = mein_fobj; // Typ: std::function<std::string(int)>
  std::cout << fobj(4) << std::endl;
...
```

UNIVERSITÄT BONN

Funktionen zur Laufzeit definieren

In modernen objektorientierten Programmiersprachen können Funktionen zur Laufzeit definiert werden. Diese Funktionen sind der Wert einer sogenannten **Lambdaexpression** und haben somit keinen Namen. Deshalb werden Sie auch **Anonyme Funktionen** oder **Lambdafunktionen** genannt.

UNIVERSITÄT BONN

Motivation

Wir hätten gern das folgende Funktionsobjekt.

```
class OHNENAMEN { // Wir sind am Namen des Funktionsobjekts nicht interessiert
private:
  const TYPCAP1 cap1; // Wir wollen dem Funktionsobjekt "Hintergrundwerte" zuweisen,
  const TYPCAP2 cap2; // die beim Aufruf von operator() benutzt werden können.
  ...
public:
  OHNENAMEN(TYPCAP1 _cap1, TYPCAP2 _cap2, ...) : cap1(_cap1), cap2(_cap2), ... {/* KEIN CODE */}
  // Implementierung von operator()
  RUECK operator() (TYPPAR1 par1, TYPPAR2 par2, ...) { CODE }
};

...
  std::function<....> f = OHNENAMEN(cap1, cap2, ...); // Erstelle Funktionsobjekt
...
```

Wir erreichen (methodisch) dasselbe Ergebnis wie folgt.

```
std::function<....> f = [cap1, cap2, ...] (TYPPAR1 par1, TYPPAR2 par2, ...) -> REUCK { CODE };
```

UNIVERSITÄT BONN

Beispiel I

Wir betrachten ein simples Beispiel.

```
int main() {
  // f benennt Funktionen die int-Parameter erhalten und std::string-Werte produziert
  std::function<std::string(int)> f;
  // f speichert die folgende Lambdafunktion
  f = [] (int x) -> std::string { return std::to_string(x*3.141592); };
  // Die durch f gespeicherte Lambdafunktion wird aufgerufen
  std::cout << f(10) << std::endl; // Druckt '31.41592'
  // f speichert die folgende Lambdafunktion mit einem Capture
  double faktor = userinput();     // Wir lesen eine Gleitkommazahl ein
  f = [faktor] (int x) -> std::string { return std::to_string(x*faktor); };
  // Die durch f gespeicherte Lambdafunktion wird aufgerufen
  std::cout << f(10) << std::endl; // Druckt das Ergebnis der Berechnung 10*faktor
}
```

UNIVERSITÄT BONN

Beispiel II

Es kommt immerwieder vor, dass `std::sort` eine `compare`-Funktion als Lambdaexpression übergeben wird.

```cpp
#include <vector>
#include <string>
#include <algorithm>
#include <fmt/core.h>
#include <fmt/ranges.h>

int main() {
  std::vector<std::string> coole_strings {"Hi", "Ada", "Lovelace", "ist", "cool"};
  fmt::print("Strings: {}\n", coole_strings);
  std::sort(
    coole_strings.begin(),
    coole_strings.end(),
    [](const std::string& x, const std::string& y) -> bool { return x < y; }
  );
  fmt::print("Strings: {}\n", coole_strings);
}
```

UNIVERSITÄT BONN

Beispiel III

Aus der Mathematik kennen wir die Ableitungsfunktion:

$$\frac{d}{dt} : C^\infty(\mathbb{R},\mathbb{R}) \to C^\infty(\mathbb{R},\mathbb{R}) \qquad \frac{df}{dt}(x) = \lim_{t\to 0} \frac{f(x+t)-f(x)}{(x+t)-x}$$

```
std::function<double(double)> ableiten(std::function<double(double)> f) {
    std::function<double(double)> dfdt  =
      [f] (double x) -> double {
        return (f(x+1e-5) - f(x))/1e-5;
      };
    return dfdt;
}
...
  std::function<double(double)> g = [] (double x) -> double {return x*x;};
  fmt::print("{} {} {}\n", g(1), g(2), g(3));
  std::function<double(double)> gstrich = ableiten(g);
  fmt::print("{} {} {}\n", gstrich(1), gstrich(2), gstrich(3));
...
```

Unsere Funktion `ableiten` dient hier nur zur Demonstration. Für numerische Berechnungen ist sie ungeeignet.

Haben Sie Fragen?

Exkurs: Funktionsobjekte

Weiterführendes

Lambdafunktionen

Captures werden üblicherweise als **Capture by Const Value** übergeben. Die Übergabe als **Capture by Const Value** ist im Normalfall genau das Gewünschte. Zum Einen verhält sich die Funktion bei jedem Aufruf identisch und zum Anderen ist sichergestellt, dass der Wert existiert auch wenn die Lambdafunktion selbst als Rückgabewert zurückgegeben wird (und damit den Scope verlässt indem Sie erzeugt wurde).

Es ist möglich die Captures als **Capture by Value**, **Capture by Const Reference** und **Capture by Reference** zu übergeben. Hier ist Vorsicht geboten. Jede Variante kann zu unerwarteten Fehlern oder unerwartetem Verhalten führen.

UNIVERSITÄT BONN

Funktionsparameter binden

Gegeben eine mehrstellige Funtion `f(p1, p2, p3, ..., pn)` erhalten wir daraus wie folgt neue Funktionen.

- **Parameterumordnung**: Falls es die Parametertypen erlauben, können diese umgeordnet werden oder auch mehrfach eingesetzt werden.
- **Parameterfestlegung**: Parameter werden durch Konstanten oder Referenzen ersetzt.
- **Kombinationen** der oben genannten Ansätze.

Man spricht hierbei von **Parameterbindung**.

In C++ ist es möglich Parameter zu binden. Hierzu werden die Funktionen `std::bind`[2], `std::ref` und `std::cref`[3] sowie `std::placeholders`[4] verwendet.

[2] Siehe https://en.cppreference.com/w/cpp/utility/functional/bind

[3] Siehe https://en.cppreference.com/w/cpp/utility/functional/ref

[4] Siehe https://en.cppreference.com/w/cpp/utility/functional/placeholders

Unterschiede zwischen Funktionszeiger und Funktionsobjekten

C-Programmierer:innen sind Rawpointer auf Funktionen bekannt. Rawpointer auf Funktionen unterscheiden sich von Funktionsobjekten.

Funktionspointer speichern die Startadresse von Funktionen, welche einen passenden Rückgabetyp und passende Parametern besitzt. Sie sorgen für minimal nachvollziehbaren, wartbaren Code sowie maximale Speicher- und Laufzeiteffizienz.

Funktionsobjekte speichern zusätzlich Captures und sorgen für sehr nachvollziehareren, wartbareren Code.

Beim Entwurf von nachvollziehbarem, wartbarem Code sollen keine Rawpointer auf Funktionen verwendet werden.

UNIVERSITÄT BONN

Memberfunktionen als Funktionsobjekte auffassen I

Auch Memberfunktionen eines Objekts können als Funktionsobjekte aufgefasst werden. Hierbei muss man unterscheiden, ob man über die Memberfunktion an sich oder die Memberfunktion eines ausgewählten Objekts sprechen möchte.

UNIVERSITÄT BONN

Memberfunktionen als Funktionsobjekte auffassen II

Im ersten Fall kann man die Memberfunktion an sich als Funktionsobjekt speichern. Beim Aufruf muss dann immer ein zugehöriges Objekt mitgenannt werden (auf dem die Memberfunktion ausgeführt wird). Der erste Parameter ist deshalb immer eine Objektreferenz.

```
class MeineKlasse {
public:
  MeineKlasse(int d) : data(d) {}
  void tue_dinge(int x) { std::cout << x + data << std::endl;

private:
  int data;
};

...
  std::function<void(MeineKlasse&, int)> tue_es = std::mem_fn(&MeineKlasse::tue_dinge);
  MeineKlasse k(42);
  tue_es(k, 9);
...
```

UNIVERSITÄT BONN

Memberfunktionen als Funktionsobjekte auffassen III

Im zweiten Fall kann man die Memberfunktion eines festen Objekts Funktionsobjekt speichern. Dazu wird die Referenz auf das Objekt in der Memberfunktion gebunden.

```
class MeineKlasse {
public:
  MeineKlasse(int d) : data(d) {}
  void tue_dinge(int x) { std::cout << x + data << std::endl;

private:
  int data;
};

...
  MeineKlasse k(42);
  std::function<void(int)> tue_es_anders;
  tue_es_anders = std::bind(&MeineKlasse::tue_dinge, std::ref(k), std::placeholders::_1);
  tue_es_anders(42);
...
```

Haben Sie Fragen?

Zusammenfassung

Es gibt Funktionsobjekte und jede Funktion kann als ein solches aufgefasst werden

Funktionen werden als Funktionsobjekte übergeben

In C++ ist `std::function` eine universelle Schnittstelle um Funktionen und Funktionsobjekte zu behandeln

Abstrakte Datentypen und Datenstrukturen

Wo und wie gibt man eine totale Ordnung oder eine Hashfunktion an?

Offene Frage

Wo und wie gibt man eine totale Ordnung oder eine Hashfunktion an?

UNIVERSITÄT BONN

Geodnete Multimengen in C++ implementieren?

Gegeben ein fester Typ T, deklarieren wir nun einen Datentyp, der geordnete Multimengen von T-Objekten realisiert.

```
class GeordneteMultimengeVonT {
public:
  GeordneteMengeVonT(std::function<bool(const T&, const T&)> compare_fkt)
    : cmp_fkt(compare_fkt) // Instanziierung
  { ... }

  ...
private:
  const std::function<bool(const T&, const T&)> cmp_fkt; // Nach Instanziierung unveränderbar
}
```

Natürlich wollen wir geordnete Multimengen für beliebige Typen deklarieren und implementieren. Unsere Implementierung wird so sein, dass der gewählte Typ austauschbar ist. Wie man eine Implementierung unabhängig vom Datentyp vornimmt, lernen wir später, beim Einstieg in die **generische Programmierung**.

UNIVERSITÄT BONN

Hashende Multimengen in C++ implementieren?

Gegeben ein fester Typ T, deklarieren wir nun einen Datentyp, der geordnete Multimengen von T-Objekten realisiert.

```
class HashendeMengeVonT {
public:
  HashendeMengeVonT(std::function<uint64_t(const T&)> hash_fkt)
    : hash(hash_fkt) // Instanziierung
  { ... }

  ...
private:
  const std::function<uint64_t(const T&)> hash; // Nach Instanziierung unveränderbar
}
```

Natürlich wollen wir geordnete Multimengen für beliebige Typen deklarieren und implementieren. Unsere Implementierung wird so sein, dass der gewählte Typ austauschbar ist. Wie man eine Implementierung unabhängig vom Datentyp vornimmt, lernen wir später, beim Einstieg in die **generische Programmierung**.

Haben Sie Fragen?

Zusammenfassung

Man übergibt totale Ordnungen oder Hashfunktionen als Funktionen

Funktionen werden als Funktionsobjekte übergeben

In C++ ist `std::function` eine universelle Schnittstelle um Funktionen und Funktionsobjekte zu behandeln

Abstrakte Datentypen und Datenstrukturen

Abstrakte Datentypen und Datenstrukturen in C++

Offene Frage

Wie realisiert man abstrakte Datentypen in C++?

Abstrakte Datentypen und C++

Erinnerung: Es gibt viele, ähnliche Definition von *den abstrakten Datentypen*. Allerdings gibt es keine allgemein anerkannte Definition. Jedes Gebiet und jede Programmiersprache legt (oft nicht und manchmal nur zum Teil) fest, wie dort die abstrakten Datentypen definiert sind.

In C++ wurden abstrakte Datentypen bis zu C++20 nicht explizit definiert. Seit C++20 wird ein Teil der abstrakten Datentypen durch `concepts` beschrieben.

UNIVERSITÄT BONN

Datenstrukturen in C++ die abstrakte Datentypen realisieren I

In C++ sind Datenstrukturen definiert, welche die hier genannten abstrakten Datentypen vollständig (oder wenigstens fast vollständig) realisieren.

Achtung: Die Benennung der Datenstrukturen ist anfangs verwirrend. Es ist auf den ersten Blick verwirrend, dass diese Datenstrukturen denselben Namen tragen wie die zugehörigen abstrakten Datenstrukturen die sie realisieren. Mit diesem Umstand müssen wir sowohl in C++ und auch in anderen Programmiersprachen leben lernen.

UNIVERSITÄT BONN

Datenstrukturen in C++ die abstrakte Datentypen realisieren II

Abstrakter Datentyp	Datenstruktur
Stack	`std::stack`
Queue	`std::queue`
Liste	`std::forward_list` bzw. `std::list`
Array	`std::vector`
Priority Queue	`std::priority_queue`
geordnete Menge	`std::set` bzw. `std::multiset`
geordnete Abbildung	`std::map` bzw. `std::multimap`
hashende Menge	`std::unordered_set` bzw. `std::unordered_multiset`
hashende Abbildung	`std::unordered_map` bzw. `std::unordered_multimap`

Vergleichen Sie auch `https://en.cppreference.com/w/cpp/container`

Haben Sie Fragen?

Zusammenfassung

In C++ gibt die Standardbliothek Datenstrukturen vor, welche die hier vorgestellten abstrakten Datentypen realisieren

Für eine eigene Implementierung müssen wir erst lernen, wie die Prinzipien der generische Programmierung in C++ umgesetzt werden

Abstrakte Datentypen und Datenstrukturen

Zusammenfassung

UNIVERSITÄT BONN

Zusammenfassung

Sie haben eine Vielzahl an abstrakten Datentypen kennen gelernt. Die hier vorgestellten sind allesamt Container. Die Container unterscheiden sich anhand der möglichen Interaktionen und der Interaktionskosten.

Um einen geeigneten Container zu wählen, bestimmen Sie alle nötigen Interaktionen mit diesem Container und wählen anschließend den Container aus, der diese Interaktionen ermöglich und dabei möglichst effizient ist.

Sie haben für jeden Container einen Datentyp kennen gelernt, der den Container realisiert.