C/C++ Materialpaket (Level C)
06_STD – Standard Library

Prof. Dr. Carsten Link

Zusammenfassung

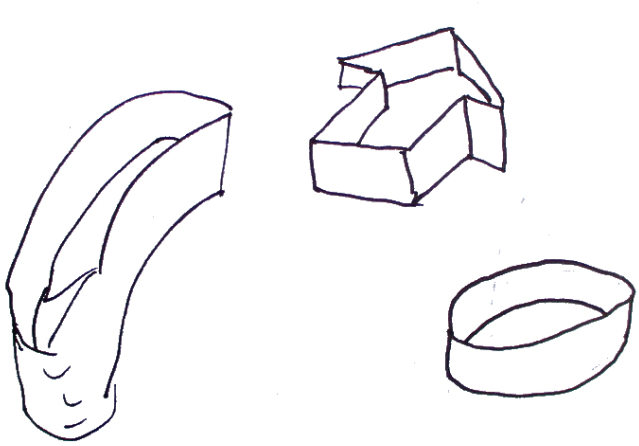

Abbildung 1: Beim Plätzchen Backen kommen Musterstanzer zum Einsatz.

Inhaltsverzeichnis

1 Kompetenzen und Lernegebnisse

Durch das Bearbeiten dieses Materialpaketes erwerben Sie diese Kompetenzen (Wissen, Fähigkeiten und Fertigkeiten zur Problemlösung):

Sie können einfache Templates implementieren und Teile der C++-Standardbibliothek verwenden.

Die oben genannten Kompetenzen erwerben Sie, indem Sie Lernziele erreichen, welche sich prüfen lassen. Lernegebnisse: Sie können nachweislich[1]:

- einfache Template-Funktionen umsetzen
- anhand eines einfachen Containers nachvollziehen, wie `template`-Klassen eingesetzt werden
- die wichtigsten Container der C++-Standardbibliothek verwenden
- einige wenige Algorithmen der C++-Standardbibliothek verwenden

2 Konzepte

Im Folgenden wird das Konzept des *parametric polymorphism* dargestellt, welches es dem Programmierer gestattet, Code zu schreiben, der mit unterschiedlichen (dennoch gleichartigen) Datentypen rechnet.

Des Weiteren werden wichtige Komponenten der C++-Standardbibliothek vorgestellt.

2.1 Generische Funktionen (Funktionstemplates)

Der Templatemechanismus erlaubt es, Funktionen vom Compiler generieren zu lassen. Programmierer formulieren in einem Funktionstemplate einen Algorithmus, der mit unterschiedlichen unbenannten Typen arbeiten kann. Der Compiler generiert bei Bedarf Ausprägungen des Templates mit den zur Compilierzeit ermittelten konkreten Typen.

2.1.1 Funktionstemplates mit Typparametern

Im Folgenden soll die Funktion `max3()` als Beispiel dienen. Sie akzeptiert drei Werte und liefert den größten davon zurück. Function Overloading erlaubt es

[1] Sie können das Erzielen der einzelnen Lernergebnisse beispielsweise bei einem Testat im Praktikum oder einer Aufgabe in der Modulprüfung nachweisen.

uns diese Funktion für mehrere verschiedene Typen zu implementieren:

```
int         max3(int a, int b, int c);
RgbColor    max3(RgbColor a, RgbColor b, RgbColor c);
std::string max3(std::string a, std::string b, std::string c);
```

Die Nachteile dieser Umsetzung mit überladenen Funktionen sind: unnötige Tipparbeit, Bugs und Features müsse mehrfach eingepflegt werden und Tests müssen mehrfach geschrieben werden.

Mit C++-Templates haben wir die Möglichkeit, eine Familie von Funktionen zu definieren. Der Compiler generiert dann bei der Übersetzung konkrete Ausprägungen der Funktionen:

```
#include "println.hpp"

template <typename T>
T max3(T a, T b, T c){
  if ( (a>b) && (a>c) ) {
    return a;
  } else if ( (b>a) && (b>c) ) {
    return b;
  } else {
    return c;
  }
}

int main(){
  int i1=1, i2=2, i3=3;
  int mi = max3(i1,i2,i3);
  println("max3(i1,i2,i3) == ", mi);
  double d1=1.0, d2=2.0, d3=3.0;
  double md = max3(d1,d2,d3);
  println("max3(d1,d2,d3) == ", md);
  return 0;
}
```

In obigem Quellcode wird der Compiler die `max3`-Funktionen für die Typen `int` und `double` generieren.

2.1.2 (Level C) Funktionstemplates mit Wertparametern

In Materialpaket 03b_DATA_rep wurden Funktionen erarbeitet, die Zahlen zwischen `int` und `std::string`-Darstellung (und umgekehrt) umwandeln. Da die Funktionen `numberToHexString()` und `numberToDecString()` im wesentlichen denselben Code haben, bietet es sich an, den gemeinsamen Code in eine neue Funktion auszulagern und zwei convenience-Funktionen anzubieten, um die alte Schnittstelle zu erhalten:

```
std::string numberToHexString(unsigned long value){
  return numberToAsciiString(value, 16);
}

std::string numberToDecString(unsigned long value){
  return numberToAsciiString(value, 10);
}
```

So ist der Code zur Umwandlung an einer zentralen Stelle gut wartbar; auch die Umwandlung weiterer Zahlensysteme lässt sich leicht umsetzen. Ein Nachteil ist jedoch, dass nun Funktionsaufrufe nötig sind, was CPU-Zyklen und damit Zeit und Energie beansprucht. Da die Basen der Zahlensysteme (`16` und `10`) zur Laufzeit bekannt sind, bietet sich hier ein Funktionstemplate mit `unsigned long`-Parameter an:

```
template<unsigned long BASE>

std::string numberToAsciiString_T(unsigned long value){
  std::string result = "";
  do {
    unsigned long lastDigit  = value % BASE;            // last digit`s int value
    char c = numberToAsciiDigit(lastDigit); // last digit`s character
    value =  value / BASE;                     // all remaining digits, i.e. shift right
    result += c;
  } while (value);
  std::reverse(result.begin(), result.end());
  return result;
}
```

Der Aufruf der Funktionerfolgt beispielsweise so:

```
  for (unsigned long i=0; i<iterations;i++){
   println(i, " dec = hex ", numberToAsciiString_T<16>(i));
  }
```

Im Verzeichnis `src-cpp-student/07_STD` ist mit den Dateien `z_int2str*` und `z_build_int2str.sh` nachvollziebar, dass das Programm, das die Templatefunktion verwendet, doppelt so schnell ist, wie das, welches die Basisauswahl zur Laufzeit verwendet.

2.1.3 println()-Funktionstemplate

In früheren Materialpaketen wurde die Funktion `println()` vorgestellt, mit der Ausgaben gemacht werden können. Hierbei ist es möglich, diese Funktion mit Parametern verschiedenen Typs zu verwenden. Letztlich handelt es sich bei `println()` um eine überladene Funktion – jedoch wurden die jeweiligen Definionen dafür vom Compiler generiert: `println()` ist ein Funktionstemplate

in `println.hpp`:

```
template <typename T>
void println(T t)
{
  std::cout << as_string(t) << std::endl;
}
```

Aufgrund von `template <typename T>` übersetzt der Compiler die Funktion nicht wie gewöhnliche Funktionen, sondern generiert bei Bedarf `println(T)`-Überladungen für die im Quelltext verwendeten Typen `T`. Im folgenden Beispiel werden vom Compiler drei verschiedene Überladungen erzeugt (für die Typen `double`, `int`, und `RationalNumber`):

```
double myPi = 355.0/113.0;    println(myPi); //println(double);
int i=42;                     println(i);    //println(int);
RationalNumber pir(355, 113); println(pir);  //println(RationalNumber);
```

Die Template-Funktion `println()` ist mittels `as_string()` implementiert, wovon Spezialisierungen für die eingebauten Typen in `println.hpp` gegeben sind; Spezialisierungen von `as_string()` für benutzerdefinierte Typen müssen vom Programmierer der jeweiligen Typen gegeben werden (wie beispielsweise bei `RationalNumber`, `PascalStr` und `RgbColor`).

(Level C) In `println.hpp` ist zu sehen, wie es ermöglicht wird, dass die Templatefunktion `println()` beliebig viele Parameter akzeptiert.

2.2 Ein Stack für int-Werte

Die vielfach verwendete Datenstruktur Stack soll anhand einer einfachen Implementierung für `int`-Werte illustriert werden. Zunächst die Typdefinition:

```
#ifndef intStack_hpp
#define intStack_hpp

// good: encapsulation
// good: separation of interface and implementation
// bad: fixed element type
// bad: fixed capacity

struct intStack {
  static const int capacity=1024;
  int tos; // top of stack, i.e. index of next write/push
  int elements[capacity];
public:
  intStack();
  void push(int element);
  int pop();
```

```
17    int size(); // number of elements pushed
18    bool isEmpty();
19    void clear();
20    void print();
21  };
22
23  #endif
```

Diese Klasse stellt einen Werttypen dar, in dem bis zu 1024 `int`-Werte gespeichert (`push()`) und wieder abgefragt (`pop()`) werden können. Ein Zugriff auf beliebige Stellen ist nicht vorgesehen. Die Membervariable `tos` speichert den Index der beim nächsten `push()` zu beschreibenden Zelle im Array `elements` (beim Beschrieben wird `tos` inkrementiert). In den Kommentaren in obigem Quelltext sind wesentliche Nachteile dieser einfachen Implementierung zu sehen:

- der Typ der Elemente ist auf `int` festgelegt
- die Kapazität ist auf `1024` festgelegt

Die folgende Funktion `use_intStack()` zeigt, wie ein Stapel verwendet werden kann:

```
1   void use_intStack(){
2     intStack s_1;
3     for(int i=0; i<10;i++){
4       s_1.push(i*i);
5     }
6     std::cout << "s_1 : " << std::endl;
7     s_1.print();
8     // replace top element by 4711:
9     s_1.pop();
10    s_1.push(4711);
11    // swap topmost two elements:
12    int tmp1 = s_1.pop();
13    int tmp2 = s_1.pop();
14    s_1.push(tmp1);
15    s_1.push(tmp2);
16    std::cout << "draining s_1 : " << std::endl;
17    while(!s_1.isEmpty()){
18      std::cout << s_1.pop() << std::endl;
19    }
20  }
```

In obigem Quelltext werden die Quadratzahlen von `0..81` auf einem Stack gespeichert. Das oberste Element wird durch `4711` getauscht, wozu eine `pop()/push()`-Kombination notwendig ist. Das anschließende Tauschen der obersten beiden Elemente benötigt sogar zwei lokale Variablen (`tmp1` und `tmp2`).

Die Ausgabe von `use_intStack()` sieht wie folgt aus:

```
use_intStack()
--------------
s_1 :
10 of 1024 allocated.
0: 0
1: 1
2: 4
3: 9
4: 16
5: 25
6: 36
7: 49
8: 64
9: 81
draining s_1 :
64
4711
49
36
25
16
9
4
1
0
```

2.3 Ein generischer Stack (Klassentemplate)

Der folgende Quellcode verdeutlicht, wie Templates helfen können, die Probleme des `intStack` zu vermeiden: Es lassen sich `genericStack`-Objekte anlegen, welche verschiede Elementtypen erlauben und unterschiedliche feste Größen haben (`genericStack<int, 20>` und `genericStack<double, 30>`).

```
1  void use_genericStack(){
2    genericStack<int, 20> s_2;
3    genericStack<double, 30> s_3;
4    for(int i=0; i<10;i++){
5      s_2.push(i*i);
6      s_3.push(i*i * 1.1);
7    }
8    s_2.print();
9    s_3.print();
10 }
```

Die Ausgabe von `use_genericStack()` sieht wie folgt aus:

```
use_genericStack()
------------------
10 of 20 allocated.
0: 0
1: 1
2: 4
3: 9
4: 16
5: 25
6: 36
7: 49
8: 64
9: 81
10 of 30 allocated.
0: 0
1: 1.1
2: 4.4
3: 9.9
4: 17.6
5: 27.5
6: 39.6
7: 53.9
8: 70.4
9: 89.1
```

Die Definition der Template-Klasse `genericStack`:

```
1  //  genericStack.hpp
2
3  #ifndef genericStack_hpp
4  #define genericStack_hpp
5  #include <iostream>
6
7  // good: encapsulation
8  // good: separation of interface and implementation
9  // good: arbitrary element type
10 // good: arbitrary capacity
11 // bad: implementation in .hpp
12 // bad: compiler error messages are difficult to read
13
14 template <typename ElementType, int capacity>
15 struct genericStack {
16   int tos; // top of stack, i.e. index of next write/push
17   ElementType elements[capacity];
18 public:
19   genericStack();
```

```
20    void push(ElementType element);
21    ElementType pop();
22    int size(); // number of elements pushed
23    bool isEmpty();
24    void clear();
25    void print();
26  };
```

Die Methoden sind analog zu denen des `intStack`s implementiert. Die wesentlichen Unterschiede sind jedoch:

- die Methoden müssen in der Header-Datei vollständig implementiert sein, da der Compiler diese Definitionen an den benutzenden Stellen benötigt
- bei jeder Methodendefinition müssen die Template-Parameter angeführt sein (`template <typename ElementType, int capacity> ... <ElementType, capacity>`)

```
1   //  genericStack.hpp
2   // ============= implementation =============
3
4   template <typename ElementType, int capacity>
5   genericStack<ElementType, capacity>::genericStack()
6   : tos(0)
7   {
8
9   }
10
11  template <typename ElementType, int capacity>
12  void genericStack<ElementType, capacity>::push(ElementType element){
13    elements[tos++] = element;
14  }
15
16  template <typename ElementType, int capacity>
17  ElementType genericStack<ElementType, capacity>::pop(){
18    return elements[--tos];
19  }
20
21  template <typename ElementType, int capacity>
22  int genericStack<ElementType, capacity>::size(){ // number of elements pushed
23    return tos;
24  }
25
26  template <typename ElementType, int capacity>
27  bool genericStack<ElementType, capacity>::isEmpty(){
28    return tos == 0;
29  }
```

```
30
31 template <typename ElementType, int capacity>
32 void genericStack<ElementType, capacity>::clear(){
33   tos = 0;
34 }
35
36 template <typename ElementType, int capacity>
37 void genericStack<ElementType, capacity>::print(){
38   std::cout << size() << " of " << capacity << " allocated." << std::endl;
39   for (int i=0; i<tos; i++){
40     std::cout << i << ": " << elements[i] << std::endl;
41   }
42 }
43 #endif
```

2.4 Wichtige Container der C++-Standardbibliothek

- `std::vector`
- `std::list`
- `std::deque`
- `std::map`

Die Standardcontainer unterscheiden sich hinsichtlich der Zugriffsmöglichkeiten und der zu erwartenden Laufzeit einzelner Operationen (so kann erwartet werden, dass das Einfügen in Listen effizienter vonstatten geht, als das Einfügen in Vektoren).

Der folgende Code zeigt die Verwendung der beiden Standardcontainer `std::vector` und `std::list`:

```
1  void use_std_containers(){
2    std::vector<int> v;
3    v.push_back(17);          // append an element at the end
4    int i = v.back();         // get last element
5    int j = v[0];             // access by subscript operator
6    if ((i==j) && (i==17)){   //
7      v.pop_back();           // remove last element
8      v.clear();              // remove all elements
9    }
10   long s = v.size();        // get size; size() <= max_size()
11   std::list<long> l;
12   l.push_back(17);          // append an element at the end
13   l.push_front(s);          // put an element in front of the list
14 }
```

Es wird ersichtlich, dass es nicht notwendig ist, eine eigene Stack-Klasse (wie `genericStack`) zu implementieren, da die Standardcontainer die notwendigen

Operationen mitbringen.

Siehe hierzu auch [PPP] Seite 1146.

2.5 Wichtige Algorithmen der C++-Standardbibliothek

Eine vollständige Auflistung der Algorithmen der C++-Standardbibliothek findet sich hier[2]. Hier sollen nur einige wenige Algortihmen vorgestellt werden:

- `std::sort`
- `std::transform`

Der folgende Quellcode zeigt die Benutzung dieser beiden Standardalgorithmen:

```
1  void use_std_algorithms(){
2    std::vector<int> v(100);
3    for(int i=0; i<10; i++){
4      v[i*i] = i;
5    }
6    v.push_back(17);
7    printVector(v);
8    std::sort(v.begin(), v.end());
9    printVector(v);
10   std::transform(v.begin(), v.end(), v.begin(),
11                  [](int i) { return i*10; }); // use lambda expression here
12   printVector(v);
13 }
```

An der Ausgabe ist zu erkennen, wie der Inhalt des Vektors sortiert und schließlich mit `10` multipliziert wird:

```
// some output omitted
[0]=0, [1]=0, [2]=0, [3]=0, [4]=0, [5]=0, [6]=0, [7]=0, [8]=0,
[9]=0, [10]=0, [11]=0, [12]=0, [13]=0, [14]=0, [15]=0, [16]=0,
// ...
[73]=0, [74]=0, [75]=0, [76]=0, [77]=0, [78]=0, [79]=0, [80]=0,
[81]=0, [82]=0, [83]=0, [84]=0, [85]=0, [86]=0, [87]=0, [88]=0,
[89]=0, [90]=0, [91]=10, [92]=20, [93]=30, [94]=40, [95]=50,
[96]=60, [97]=70, [98]=80, [99]=90, [100]=170,
```

3 Material zum aktiven Lernen

Regelmäßiger Hinweis: Da eine Programmiersprache nur durch aktive Verwendung erlernt werden kann, werden im Folgenden Aufgaben zum praktischen Üben vorgestellt. Zunächst wird ein Grundgerüst (C/C++Programm) erstellt,

[2] http://en.cppreference.com/w/cpp/algorithm

welches dann auf mehrere Arten modifiziert wird. Insbesondere die Modifikationen ermöglichen es dem Lernenden (und auch dem Lehrenden), die Qualität des Kompetenzerwerbs bzgl. dieses Materialpakets bewerten zu können.

3.1 Aufgabe: Grundgerüst

Achten Sie darauf, bei den Projekteinstellung den C++14-Standard zu aktivieren (unter Workspace - active project settings - global settings - C++ compiler options - ...).

Level C: Schreiben sie eine Funktion `double stringSimilarity()`, die für zwei `std::string` auf einer Skala von `0.0` bis `1.0` zurückgibt, wie ähnlich die beiden Strings sind. Der Wert `1.0` ist für vollständig gleiche Zeichenketten zurückzugeben. Zeigen Sie die Funktionsfähigkeit von `stringSimilarity()` anhand einiger Testaufrufe in `void unittest_stringSimilarity()`.

Überlegen Sie, wie Sie die beiden Strings mit traditionellen Mitteln (Iteration, Zugriff per Subscript Operator `[ ]`, etc.) vergleichen können, wobei der Anteil gleicher Zeichen ein wesentlicher Aspekt ist – unabhängig von deren Position.

Für die endgültige Lösung sollten Sie nun Funktionen aus der Standardbibliothek verwenden, beispielsweise: `std::transform()` mit `std::toupper()`, `std::sort()` und `std::unique()` zur Normalisierung der zu vergleichenden Strings sowie `std::string.length()`und `std::max()` zur Berechnung.

Schauen Sie sich auch `std::set_intersection()` an – damit läßt sich womöglich eine sehr kurze Lösung zur Berechnung des Ähnlichkeitswertes implementieren.

Nehmen Sie den unten gegebenen Quellcode sowie Ihre `RgbColor`-Klasse aus dem Materialpaket 04_UDEF als Basis. Generieren Sie `42` `int`-Zufallswerte aus dem Bereich `0..23` und füllen damit einen `std::vector<RgbColor>` (dazu benötigen Sie einen geeigneten Konstruktor in der Klasse `RgbColor`).

```
1  #include <ctime>
2  #include <vector>
3  #include <list>
4  #include <algorithm>
5  #include "println.hpp"
6  #include "RgbColor.hpp"
7
8  int main()
9  {
10   std::vector<RgbColor> fixedNums;
11   std::srand(std::time(0)); // use current time as seed for random generator
12   fixedNums.push_back(std::rand());
13 }
```

Wenn Sie den Aufruf von `std::srand()` weglassen, erleichtert sich die Fehlersuche, da Sie dann in jedem Lauf die selbe Sequenz von Zufallszahlen erhalten.

Hinweis: wenn Sie dem `#include "println.hpp"` ein `#include "cxx-prettyprint/prettyprint.hpp"` voranstellen, lassen sich auch Container an `println()` übergeben.

3.2 Aufgabe: Modifikationen

Regelmäßiger Hinweis: Weiter unten ist eine Liste mit Modifikationen gegeben, die zwei Zwecken dienen: 1) Sie dienen als Richtschnur für das Praktizieren und Üben der Inhalte dieses Materialpakets. 2) Die Modifikationen können im Rahmen eines Testats als Aufgabe verwendet werden, durch deren Lösung Studierende nachweisen können, dass sie den Stoff dieses Materialpakets beherrschen. Stellen Sie sicher, dass Sie jede einzelne der nachfolgenden Modifikationen innerhalb weniger Minuten (ca. 5 - 10) vor Zuschauern (Testatsituation) umsetzen können. Konkret sollen Sie im Testat in der Lage sein, das gegebene Grundgerüst um mindestens eine zufällig ausgewählte Modifikation zu erweitern. Bereiten Sie dazu auf ihrer Arbeitsumgebung ein Verzeichnis vor, welches ausschließlich das Grundgerüst enthält. **Arbeiten Sie also auf einer Kopie des Verzeichnisses `src-cpp-student/`!** Achten Sie darauf, dass der Text auf Ihrem Bildschirm in heller Umgebung aus einem Meter Abstand heraus gut lesbar ist (light mode, große Schrift).

Modifikationen:

1. Mit Zufallszahlen initialisierte `RgbColor`-Werte in einen `vector<>` ohne Duplikate einfügen
2. Funktion schreiben, welche den gefüllten `vector<RgbColor>` CSV-artig ausgibt (`"7", "9", "1", ...`)
3. Mit Zufallszahlen initialisierte `RgbColor`-Werte in `list<>` ohne Duplikate einfügen (`std::find` darf verwendet werden)
4. Mit Zufallszahlen initialisierte `RgbColor`-Werte in `vector<>` nacheinander an die richtige Stelle (sortiert) einfügen
5. Mit Zufallszahlen initialisierte `RgbColor`-Werte in `list<>` nacheinander an die richtige Stelle (sortiert) einfügen
6. Mehrere `RgbColor` in einen `vector<>` speichern und dort von `std::sort()` sortieren lassen
7. **Level C:** Ändern Sie Logik in `stringSimilarity()`, so dass die Groß-/Kleinschreibung berücksichtigt wird und eine andere Skala verwendet wird.
8. **Level C:** stellen Sie `printMinMax()` aus der Datei `main_04_UDEF_minmax.cpp` aus dem Materialpaket 04_UDEF auf eine Template-Funktion um, so dass von `main()` aus alle vier Varianten direkt nacheinander aufgerufen werden können (ohne `#ifdef` und ohne `typedef`)
9. **Level C:** nehmen Sie Ihren Code der `convertInto()`-Funktionen und stellen diesen eine Template-Funktion voran, so dass komfortabel per `convert<T>(e)` ein Ausdruck `e` der Typen `double`, `int`, `std::string` in einen Wert vom Typ `T` umgewandelt werden kann (jeweils ein anderer Typ aus den genannten)

3.3 Verständnisfragen

Nach Bearbeitung des Kapitels "Konzepte", der Erstellung des Grundgerüsts sowie dem Üben der Modifikationen sollten Sie in der Lage sein, die folgenden Fragen zu beantworten.

1. Worin liegen die wesentlichen Unterschiede zwischen `std::vector` und `std::list`?
2. Wie gelingt es `std::sort()` einen Container zu sortieren, in dem sich Objekte befinden, welche Instanzen von Typen sind, die `sort()` nicht bekannt sind?
3. Welchen Vorteil hat die Nutzung von Container-Templates aus der Standardbibliothek gegenüber einer Eigenimplementierung?
4. Sie benötigen für ein Programmierproblem eine Datenspeicher, welche sich wie ein Stapel verhält. Wie lösen Sie dieses Problem?
5. Ein `char` kann implizit in ein `int` umgewandelt werden. Kann eine `std::list<char>` in eine `std::list<int>` implizit umgewandelt werden? Begründung!
6. Freiwillig: Zeigen Sie auf, inwiefern das Piktogram auf der Titelseite dieses Materialpaketes den Inhalt zusammengefasst darstellt.

4 Nützliche Links

- Wikibooks: C++-Programmierung, Die STL Container[3]
- Wikibooks: C++-Programmierung, Die STL Algorithmen[4]
- Hacking C++, Standard Library Algorithms https://hackingcpp.com/cpp/std/algorithms.html
- Hacking C++, Standard Algorithms Intro https://hackingcpp.com/cpp/std/algorithms/intro.html
- Hacking C++, Standard Library Containers https://hackingcpp.com/cpp/std/containers.html
- Hacking C++, Standard Library Sequence Containers https://hackingcpp.com/cpp/std/sequence_containers.html
- cxx-prettyprint: A C++ Container Pretty-Printer[5]
- The C++ Standard Library - A Tutorial and Reference[6]
- Carlos Moreno: An Introduction to the Standard Template Library (STL)[7]
- Chua Hock-Chuan: C++ Standard Libraries and Standard Template Library (STL)[8]
- Learn C++: Overloading the parenthesis operator[9]

[3] https://de.wikibooks.org/wiki/C%2B%2B-Programmierung/_Die_STL/_Container
[4] https://de.wikibooks.org/wiki/C%2B%2B-Programmierung/_Die_STL/_Algorithmen
[5] http://louisdx.github.io/cxx-prettyprint/
[6] http://www.cppstdlib.com
[7] https://cal-linux.com/tutorials/STL.html
[8] https://www3.ntu.edu.sg/home/ehchua/programming/cpp/cp9_STL.html
[9] http://www.learncpp.com/cpp-tutorial/99-overloading-the-parenthesis-operator/

- Discoverable algorithms in C++, Pascal Thomet, 2020, http://code-ballads.net/discoverable-algorithms/

5 Literatur

- [PPP] Stroustrup, Bjarne: Programming - Principles and Practice using C++
- [TCPL] Stroustrup, Bjarne: The C++ Programming Language, Fourth Edition