IMS
Institut für Mikroelektronische Systeme
Leibniz Universität Hannover

Leibniz Universität Hannover

Programmierpraktikum Technische Informatik (C++)

16.6.2022

IMS
Institut für Mikroelektronische Systeme
Leibniz Universität Hannover

Leibniz
Universität
Hannover

Die Standardbibliothek: Sequentielle Container

Container der Standardbibliothek

- Zwei Containerklassen wurden bereits behandelt:
  `std::vector` und `std::map`
- Die Standardbibliothek enthält noch andere Containertypen
- Unterteilt in zwei Kategorien: sequentielle und assoziative Container
  - Unterschied: Anordnung der Elemente im Container
- Heute: Behandlung der sequentiellen Containerklassen

Sequentielle Container

- Merkmal sequentieller Container: Reihenfolge der Elemente hängt von der Einfügeposition und nicht von ihrem Wert ab
- Standardbibliothek stellt fünf sequentielle Container bereit:
  - `std::vector<T>`
  - `std::deque<T>`
  - `std::list<T>`
  - `std::forward_list<T>`
  - `std::array<T>`
- Frage: Worin unterscheiden sich die Containertypen?
- Antwort: Im Wesentlichen in ihren Performanceeigenschaften
  - Je nach Nutzungsverhalten ist mal der eine, mal der andere Container effizienter
- Performanceeigenschaften eines Containers werden über die Laufzeitkomplexität ihrer Operationen ausgedrückt

Exkurs: Laufzeitkomplexität

- Wie kann man die Performance zweier Algorithmen vergleichen?
- Ansatz: Laufzeit messen und vergleichen
- Hat einige Probleme:
  - Laufzeit hängt von dem System ab, auf dem getestet wurde
    - Ergebnisse nicht mit Messungen auf anderen Systemen vergleichbar
  - Die tatsächliche Laufzeit hängt häufig von den Eingabedaten ab
    - Über einen Vektor mit 10000 Elementen zu iterieren, dauert länger als über einen mit 10 Elementen
  - Laufzeit hängt auch von der genauen Implementierung des Algorithmus ab
- Viel interessanter als der genaue Wert ist die Größenordnung
  - In vielen Fällen sind ein paar Prozent mehr Laufzeit nicht relevant
- Betrachtung des asymptotischen Verhaltens in Abhängigkeit von der Eingabegröße

Groß-O-Notation

- Seien f und g Funktionen von $\mathbb{N}$ nach $\mathbb{N}$
- Man sagt: f ist höchstens von der Größenordnung g, wenn gilt:
  $\exists n_0, c \in \mathbb{N}\ \forall n \geq n_0 : f(n) \leq c \cdot g(n)$
  - g dominiert f
- Bedeutet, dass f asymptotisch nicht größer als g ist
- Notation: $f \in O(g)$ oder auch $f = O(g)$
  - $O(g)$ (“O von g”) ist die Menge der Funktionen, deren Größenordnung höchstens g ist

Beispiel Groß-O-Notation

- Sei $f(n) = 10 \cdot n$ und $g(n) = n^2$
  - Wer dominiert hier wen?

	1	2	5	10	20	50
$f(n)$	10	20	50	100	200	500
$g(n)$	1	4	25	100	400	2500

- Für $n \leq 10$ gilt $f(n) \geq g(n)$
- **Aber:** Für alle $n > 10$ gilt $f(n) < g(n)$
- Mit Wahl von $n_0 = 10$ und $c = 1$ gilt also $f(n) \leq g(n)\ \forall n \geq n_0$
  - Also: $f \in O(g)$
- n_0 und c sind nicht eindeutig
  - Aussage ist für $n_0 = 1$, $c = 10$ ebenfalls korrekt

Beispiel Groß-O-Notation II

- Sei $f(n) = 10 \cdot n^3 + 5 \cdot n^2 + 10$ und $g(n) = n^3$
  - Wer dominiert hier wen?
- Für alle n gilt: $g(n) < f(n)$
  - Also: $g \in O(f)$
- Aber: $f(n) = 10 \cdot n^3 + 5 \cdot n^2 + 10 \leq 10 \cdot n^3 + 5 \cdot n^3 + 10 \cdot n^3 = 25 \cdot n^3 \; \forall n \in \mathbb{N}$
- Somit gilt auch $f(n) \leq 25 \cdot g(n) \; \forall n \in \mathbb{N}$, also $f \in O(g)$

Laufzeitkomplexität

- Laufzeitkomplexität wird in Abhängigkeit der Größe des Inputs angegeben
  - Meistens Anzahl der Eingabedaten, manchmal auch Länge der Eingabedaten
- Ein Algorithmus hat eine Laufzeitkomplexität von $O(f)$, wenn die Anzahl seiner Rechenschritte durch f dominiert wird
  - Analog werden häufig auch Aussagen zum Speicherverbrauch getroffen werden (Speicherkomplexität $O(s)$), wenn die Menge des benötigten Speichers von s dominiert wird

Laufzeitkomplexität II

- Laufzeitkomplexität hängt häufig von den genauen Eingabedaten, nicht nur von der Größe ab
- Beschränkung der Betrachtung auf drei Konfigurationen
  - Worst Case: Laufzeitkomplexität bei schlechtest möglichen Eingabedaten
  - Average Case: Durchschnittliche Komplexität über alle Eingabedaten (schwierig zu ermitteln)
  - Best Case: Komplexität im Idealfall von optimalen Eingabedaten

Laufzeitkomplexitäten

- Die meisten gängigen Algorithmen fallen in eine von wenigen Laufzeitkomplexitäten
- Führt zu einer Hierarchie von Laufzeitkomplexitäten

Funktion	Name	Beispiel
1	konstant	Zugriff auf ein Element eines `std::vector`
$log(n)$	logarithmisch	Suchen eines Elements in einer `std::map`
n	linear	Iterieren über alle Elemente eines `std::vector`
$n \cdot log(n)$	loglinear	Komplexität optimaler Sortieralgorithmen (z.B. Heapsort)
n^2	quadratisch	Komplexität einfacher Sortieralgorithmen (z.B. Bubblesort)
$n^c \vert c > 1$	polynomial	Einfache Implementierung der Matrixmultiplikation
$c^n \vert c > 1$	exponentiell	Bester Algorithmus für das Traveling-Salesman-Problem
$n!$	faktoriell	Lösen des Traveling-Salesman-Problems durch Brute-Force

Laufzeitkomplexitäten II

- Welche Auswirkungen hat die Laufzeitkomplexität auf die tatsächliche Laufzeit für große Datensätze?

Funktion	1	5	10	20	100	1000
1	$0.001s$	$0.001s$	$0.001s$	$0.001s$	$0.001s$	$0.001s$
$log(n)$	$0.001s$	$0.0017s$	$0.002s$	$0.0023s$	$0.003s$	$0.004s$
n	$0.001s$	$0.005s$	$0.01s$	$0.02s$	$0.1s$	$1s$
$n \cdot log(n)$	$0.001s$	$0.0085s$	$0.02s$	$0.046s$	$0.3s$	$4s$
n^2	$0.001s$	$0.025s$	$0.1s$	$0.4s$	$10s$	$16m\,40s$
n^4	$0.001s$	$0.625s$	$10s$	$2m\,40s$	$1d\,4h$	$2.4 \cdot 10^9 J$
2^n	$0.001s$	$0.032s$	$1s$	$17m\,28s$	$4 \cdot 10^{16} J$	$3.4 \cdot 10^{287} J$
$n!$	$0.001s$	$0.120s$	$1h$	$77094J$	$3 \cdot 10^{144} J$	$1.4 \cdot 10^{2554} J$

Konzepte der Standardbibliothek

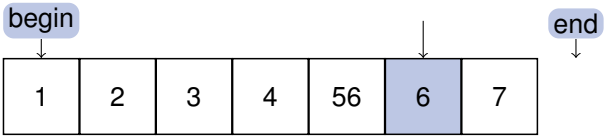
- Unterschiede zwischen sequentiellen Containern vor allem in Performancecharakteristika
- Daher auch Unterschiede in den unterstützten Operationen
  - `vector` unterstützt im Gegensatz zu `deque` und `list` kein `push_front`
  - `list` unterstützt keinen Elementzugriff über `[]` oder `.at()`
- Warum nicht alle Operationen als Member implementiert?
- Designprinzip der Standardbibliothek:
  Nur Operationen, die auf der Datenstruktur effizient implementierbar sind, sind Member
- Nicht unterstützte Operationen können häufig über andere Operationen oder Standardalgorithmen abgebildet werden

Allgemeine Laufzeitkomplexitäten

- Einige Operationen haben für alle Container identische Laufzeitkomplexitäten
- `.size()` und `.empty()` sind immer $O(1)$
- Zugriffe auf Iteratoren (`.begin()`, `.end()`, ...) sind immer $O(1)$
- Kopien von Containern sind immer $O(n)$
- `swap`- und Move-Operationen sind fast immer $O(1)$
  - Ausnahme: `std::array`

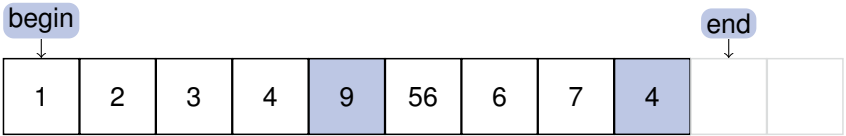
std::vector

- `std::vector` modelliert ein dynamisches Array
  - Elemente liegen in kontinuierlichem Speicher
  - Größe wird bei Einfügen angepasst
- Elementzugriff ist $O(1)$
- Problem: Änderung der Größe benötigt Allokation eines neuen (größeren) Speicherbereichs sowie Kopieren/Verschieben der Elemente in den neuen Speicher
  - Zumindest Einfügen am Ende des Vektors (gängigstes Vorgehen zum Befüllen) sollte effizienter sein



std::vector

- Lösung: Allozierter Speicherbereich ist größer als der Vektor selbst
  - `.capacity()` gibt Größe des allozierten Speichers an, `.size()` die Größe des Vektors
- Einfügen am Ende (`.push_back()` ist damit $O(1)$)
- Einfügen an anderen Positionen immer noch $O(n)$, da die Objekte nach hinten verschoben werden müssen
- Löschen hat die selben Performanceeigenschaften wie Einfügen



Laufzeitkomplexität

- `push_back` auf einem `std::vector` ist als $O(1)$ angegeben
- Aber: Nur gültig, solange noch allozierter Speicher frei ist
  - Gilt v`.capacity()==`v`.size()`, so ist kein Platz für ein weiteres Element
  - Erfordert Allokation von neuem Speicher und Verschieben/Kopieren aller Elemente des Vektors ($O(n)$)
- In gewissen Abständen ist die Laufzeitkomplexität also $O(n)$
- Effektive Laufzeitkomplexität hängt von der Häufigkeit der Reallokationen ab
- Mögliche Stategie: Immer 20 Elemente mehr als benötigt allozieren
  - Reallokation alle 20 Operationen, führt zu einer Laufzeitkomplexität von $O(n/20) = O(n)$

Armortisierte Laufzeitkomplexität

- Bessere Strategie: Größe des Speichers bei jeder Reallokation verdoppeln
  - Bei einer Größe von n erfolgt eine Reallokation in etwa nach $O(n)$ Einfügeoperationen
  - Kein Schrumpfen des Speicherbereichs beim Entfernen von Elementen
- Komplexität $O(n)$ kann nur nach jeweils n Elementen auftreten
  - Kein klassisches Worst Case Szenario
- Man sagt, die amortisierte Laufzeit ist $O(1)$
  - Gemittelte Laufzeit über gegen unendlich gehende Anzahl Operationen
  - Unterschied zur durchschnittlichen Laufzeit:
    Amortisierte Komplexität gilt auch im Worst Case

Vektor of Bool

- `std::vector<bool>` ist als sogenannte Templatespezialisierung realisiert
  - Templates können für bestimmte Argumente spezialisiert werden
    Erlaubt abweichende Implementierung für die jeweilige Argumentkombination
- Ein `bool` ist mindestens ein Byte groß
  - Kleinere Entitäten können nicht einzeln addressiert werden
- **Aber:** Ein `bool` kann nur zwei Werte annehmen
- `std::vector<bool>` verwendet für jedes Element nur ein Bit
  - Bei jedem Zugriff wird entsprechend eine Übersetzung in einen echten `bool` vorgenommen
  - Funktioniert über ein Proxyobjekt, das sich wie eine Referenz auf `bool` verhält

Vektor of Bool II

- Grundsätzlich ist eine Verringerung des Speicherbedarfs vorteilhaft
- **Aber:** Grundlegende Unterschiede zum normalen Verhalten von Vektoren
  - Keine Möglichkeit, einen Pointer oder Referenzen auf ein Element zu erhalten
- Spezialisierung von `vector<bool>` wird inzwischen als Fehlentscheidung angesehen
  - Nach Wortlaut des Standards entspricht `vector<bool>` nicht den Anforderungen eines Containers
- `vector<bool>` nur mit äußerster Vorsicht verwenden
  - Im Zweifelsfall besser über `vector<char>` emulieren
  - Alternativ: `std::deque<bool>` (später) verwenden

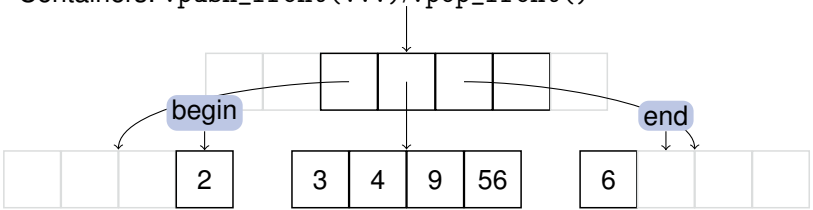
`vector<bool>` nur in Ausnahmefällen verwenden!

std::deque

- `std::vector` erlaubt effizientes Einfügen nur am Ende des Vektors
- Manchmal wird Einfügen sowohl am Anfang als auch am Ende der Sequenz benötigt
- Für dieses Szenario existiert `std::deque`
  - Effizientes Einfügen und Löschen an beiden Enden des Containers
  - Keine Garantien, dass Elemente in zusammenhängendem Speicher liegen

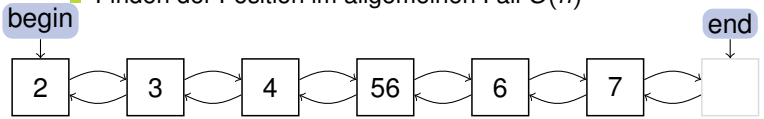
std::deque

- Übliche Implementierung: Container unterteilt in Blöcke fester Länge. Zugriff auf die Blöcke über ein Array von Pointern
- Fast alle Operationen: Gleiche Laufzeitkomplexitäten wie `std::vector`
  - Elementzugriff mit `[]` oder `.at()` in $O(1)$
  - `.push_back(..)`/`.pop_back()` ist $O(1)$ (amortisiert)
  - Einfügen/Entfernen an beliebiger Stelle ist $O(n)$
- Unterschied zum Vektor: Effizientes ($O(1)$) Einfügen/Entfernen am Anfang des Containers: `.push_front(...)`/`.pop_front()`



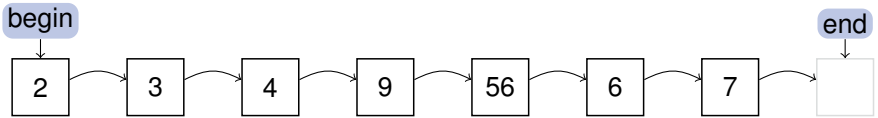
std::list

- `std::list` modelliert eine doppelt verlinkte Liste
  - Container besteht aus einzelnen Nodes, die jeweils einen Pointer zur nächsten und einen zur vorgehenden Node enthalten
- Kein wahlfreier Zugriff über `[]` oder `.at()` möglich
  - Ist für `list` nicht in $O(1)$ implementierbar
  - Zugriff auf einzelne Elemente nur über Iteratoren
- Einfügen und Löschen an beliebigen Stellen in $O(1)$, wenn man bereits einen Iterator für die Position hat
  - Finden der Position im allgemeinen Fall $O(n)$



std::forward_list

- `std::forward_list` modelliert eine einfach verlinkte Liste
  - Wie `std::list`, Nodes enthalten aber nur Pointer zu der jeweils nächsten Node
- Vorteil: Weniger Speicherbedarf als für `std::list`
- Nachteil: Eingeschränkte Zugriffsmuster
  - Keine Rückwärtsiteration möglich, mehr dazu später



std::array

- `std::array` modelliert ein statisches Array
  - Länge wird im Templateparameter angegeben, z.B. `std::array<int, 5>`
  - Nächträgliche Änderung der Länge ist nicht möglich
- Unterschied zu anderen Containern: Speicher wird inplace alloziert
  - `sizeof(std::array<`*T*`, `*N*`>)==sizeof(`*T*`)*` *N*
  - Spart dynamische Allokation
  - Vor allem bei kleinen Arrays Vorteile bzgl. der Speicherkohärenz
  - Achtung: Bedeutet, dass Move-Operationen teuer sind, da Elemente einzeln verschoben werden müssen
- Entsprechung zu C-Style Arrays (*T*`[`*N*`]`)
  - Vorteil: Verwendung des gleichen Interfaces wie für andere Container

Auswahl des richtigen Containertyps

- Der Default für sequentielle Container ist `std::vector`
  - Immer verwenden, wenn nicht **handfeste** (Performance-) Gründe dagegensprechen
- Ansonsten die benötigten Performancecharacteristika mit den Containern abgleichen und danach die Wahl treffen
- `std::deque` verwenden, wenn Einfügen an beiden Enden benötigt wird, oder für Elemente vom Typ `bool`
- `std::list` ist nur in Spezialfällen sinnvoll
- `std::forward_list` nur, wenn eine verlinkte Liste benötigt wird und der Speicherbedarf von entscheidender Bedeutung ist
- `std::array` verwenden, wenn statische Größe ausreichend und dynamische Allokation vermieden werden muss

Iteratorinvalidierung

- Bekannt: Iteratoren können durch Änderungen am Container invalidiert werden
- Mit Kenntnis der üblicherweise zugrundeliegenden Implementierung lassen sich nun genauere Aussagen treffen
- Neben Iteratorinvalidierung ist auch die Invalidierung von Pointern/Referenzen auf Elemente relevant
  - Regeln für Iteratoren und Pointer/Referenzen sind fast immer identisch

Iteratorinvalidierung II

- Invalidierung unterteilt in zwei Kategorien
  1. Speicherbereich des Elementes wurde freigegeben,
     Iterator/Referenz zeigt also ins Leere
  2. Element wurde verschoben,
     Iterator/Referenz zeigt jetzt auf ein anderes Element
- Relevante Operationen für Invalidierung:
  Alle Methoden, die potentiell die Größe ändern
  - Einfügen (`.insert(...)`, `.emplace(...)`, `.push_back(...)`, ...)
  - Löschen (`.erase(...)`, `.pop_back(...)`, ...)
  - Allgemeine Größenänderungen (`.resize(...)`, `.clear()`, ...)
    - Keine gesonderte Betrachtung notwendig,
      Verhalten genauso wie Einfügen/Löschen

Regeln für Iteratorinvalidierung

- `std::vector`
  - Einfügen/Entfernen invalidiert alle Iteratoren/Referenzen ab der Einfügeposition
  - Ist die neue Größe größer als die bisherige Kapazität, werden alle Iteratoren/Referenzen invalidiert
  - Sonst bleiben Iteratoren/Referenzen auf Objekte vor der Einfügeposition valide
- `std::deque`
  - Einfügen/Löschen invalidiert i.A. alle Iteratoren/Referenzen
  - Einfügen am Anfang/Ende invalidiert nur Iteratoren keine Referenzen
  - Löschen am Anfang/Ende invalidiert weder Iteratoren noch Referenzen
    - Ausnahme: Iteratoren/Referenzen auf gelöschte Objekte
- `std::list` und `std::forward_list`
  - Weder Iteratoren noch Referenzen werden durch Einfüge-/Löschoperationen invalidiert
    - Ausnahme: Iteratoren/Referenzen auf gelöschte Objekte

Mit Iteratorinvalidierung umgehen

- Viele Container können bei Einfüge/Löschoperationen Iteratoren invalidieren
- Problematisch für bestimmte Arten von Code
  - Insbesondere Einfügen/Löschen von mehreren Objekten an verschiedenen Stellen
- `.insert(...)` und `.erase(...)` geben häufig Iteratoren zurück
  - Rückgabewert ist ein gültiger Iterator
- Rückgabewert zeigt für Einfügeoperationen auf das eingefügte Objekt
- Löschoperationen: Rückgabewert zeigt **hinter** das gelöschte Objekt
  - Führt bei Schleifen leicht dazu, dass Elemente übersprungen werden

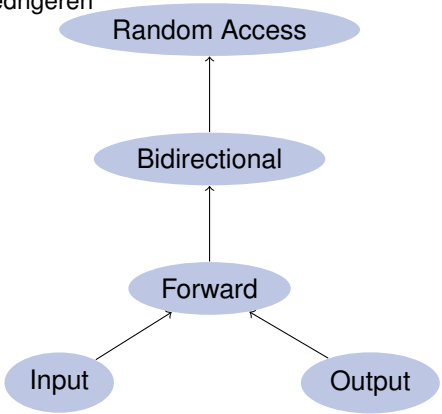
```
std::string foo;
for(auto iter = foo.begin(); iter != foo.end(); ++iter)
    if(*iter == '#')
        iter = foo.insert(iter, '\');
```

Iteratorkategorien

- Iteratoren folgen den selben Designkriterien wie Container
  - Von Iteratoren direkt unterstützte Operationen (Inkrement, Dereferenzierung, Vergleich, ...) sind immer $O(1)$
- Nicht alle Container unterstützen effizienten wahlfreien Zugriff
  - Gleiches gilt auch für Iteratoren
- Iteratoren können wie Container unterschiedliche Operationen anbieten
- Aufteilung in Iteratorkategorien

Hierarchie der Iteratorkategorien

- Iteratorkategorien lassen sich in einer Hierarchie anordnen
  - Höhere Kategorien enthalten alle Funktionen der niedrigeren
- Input- und Outputiteratoren sind Spezialfälle
  - Container unterstützen mindestens Forwarditeratoren
  - Input- und Outputiteratoren daher hier nicht behandelt
- Relevant sind also drei Kategorien: Forward, Bidirectional und Random Access



Erläuterung der Iteratorkategorien

- Forwarditeratoren können nur vorwärts durchlaufen werden
  - Unterstützt nur ++ zum Verschieben
  - Vergleiche nur mit == und `!=`, Iteratoren sind nicht geordnet
  - Iteratortyp für `std::forward_list`
- Bidirektionale Iteratoren sind wie Forwarditeratoren, unterstützen aber auch Dekrementierung (`--`)
  - Iteratortyp für alle assoziativen Container, sowie für `std::list`
- Random-Access-Iteratoren unterstützen auch wahlfreien Zugriff
  - Unterstützen dieselben arithmetischen und Vergleichsoperationen mit denselben Einschränkungen wie Pointer
  - Iteratortyp für `std::vector`, `std::deque` und `std::string`

Prev und Next

- Arithmetische Operationen auf Forward- und Bidirektional-Iteratoren ändern den Operanden
- Wird die Orginalposition noch benötigt, ist es dadurch umständlich, den Nachfolger/Vorgänger eines Iterators zu erhalten

```
auto end = map.end();
last     = end;
--last;
```

- Eleganter wäre es, `last` in einem Ausdruck zu setzen
- Für diesen Zweck existieren `std::prev` und `std::next` im Header `iterator`
  - Erhalten jeweils einen Iterator als Parameter und geben Vorgänger bzw. Nachfolger zurück

```
auto end = map.end();
last     = std::prev(end);
```

Advance und Distance

- Mit Ausnahme von Random-Access-Iteratoren können Iteratoren mit einer Operation nur um ein Element verschoben werden
  - Verschiebung um mehrere Elemente benötigt eine Schleife zur wiederholten In-/Dekrementierung
- Problematisch für generischen Code (Templates)
  - Wiederholte Inkrementierung ist $O(n)$
  - Für Random-Access-Iteratoren ist Verschiebung um mehrere Elemente mit $+$ in $O(1)$ möglich

Advance und Distance II

- Standardbibliothek definiert `std::advance` in Header `iterator`
  - `std::advance(iter, 5);`
  - Offset kann für bidirektionale Iteratoren auch negativ sein
  - Verwendet intern die optimale Implementierung für den jeweiligen Iteratortyp
- **Achtung:** Ergebnis darf nicht außerhalb des gültigen Bereichs sein!
- Analog zu `advance`: `std::distance`
  - Rückgabe: Entfernung zwischen den beiden Parametern
  - `std::ptrdiff_t dist = std::distance(map.begin(), map.end());`

Rückwärtsiteration

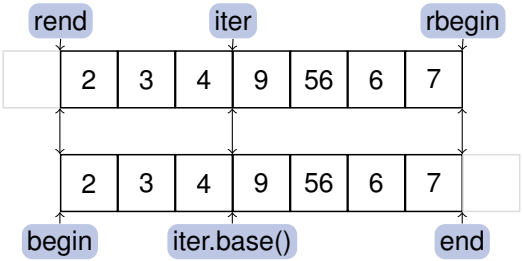
- Algorithmen der Standardbibliothek erhalten in der Regel zwei Iteratoren (Anfang und Ende des zu bearbeitenden Bereichs)
  - Wenn benötigt auch weitere Parameter
- Manchmal soll ein Algorithmus den Container in umgekehrter Reihenfolge durchlaufen
- Könnte mit einer zweiten Version aller relevanten Algorithmen gelöst werden
  - Doppelte Implementierung aller Algorithmen
  - Aufwand steigt mit der Anzahl der Algorithmen
  - Keine gut skalierende Lösung

Reverse-Iteratoren

- Die Standardbibliothek definiert Iteratoren zur umgekehrten Iteration
  - `std::reverse_iterator<`*`Iterator`*`>` im Header `iterator`
  - Container bieten in der Regel einen Typedef `reverse_iterator` und Memberfunktionen `.rbegin()` und `.rend()` an
- Reverse-Iteratoren unterstützen dieselben Operationen wie der zugrundeliegende Iteratortyp
  - Iteratorkategorie entspricht der des Basisiterators
- Arithmetische Operationen und Vergleiche werden in ihrer Richtung geändert
  - ++ ruft den -- Operator des zugrundeliegenden Iterators auf
  - Analog für +, -, <, >, <= und >=
- Auf zugrundeliegenden Iterator kann mit `.base()` zugegriffen werden

Reverse Iteratoren II

- Reverse-Iteratoren sind gegenüber dem zugrundeliegenden Iterator um eins verschoben
- Vereinfachte Merkregel: Die Position des Iterators ist jeweils zwischen den Elementen
  - Normale Iteratoren zeigen auf das jeweils nächste Element,
    Reverse-Iteratoren auf das jeweils vorgehende
  - Daher Einfügeposition in einem `vector` vor dem angezeigten Element
  - Erklärt auch, warum `.end()` hinter den gültigen Bereich zeigt



Verwendung der Iteratorkategorie

- In generischem Template-Code (z.B. `std::advance`) ist eine Unterscheidung der Iteratorkategorien sinnvoll
- Prinzipiell einfach: Iteratortypen enthalten in der Regel einen Typedef `iterator_category`
  - Element aus `std::input_iterator_tag`, `std::output_iterator_tag`, `std::forward_iterator_tag`, ...
  - Iterator-Tags bilden die Hierarchie über Vererbung nach: `std::random_access_iterator_tag` erbt von `std::bidirectional_iterator_tag` usw.
  - Tag kann intern für Overloads verwendet werden
- **Aber:** Pointer sind auch Random-Access-Iteratoren
  - Aber: T`*::iterator_category` kann nicht ausgewertet werden

Verwendung der Iteratorkategorie

- In generischem Template-Code (z.B. `std::advance`) ist eine Unterscheidung der Iterat
- Prinz
  `iter`
  - I
  - I
  - T
- **Aber**
  - A

```
template<typename Iterator>
ptrdiff_t distance_helper(Iterator begin, Iterator end,
     std::random_access_iterator_tag){
    return end - begin;
}
template<typename Iterator>
ptrdiff_t distance_helper(Iterator begin, Iterator end, std::forward_iterator_tag){
    ptrdiff_t result = 0;
    for(; begin != end; ++begin)
        ++result;
    return result;
}
//more overloads if needed

template<typename Iterator>
ptrdiff_t distance(Iterator begin, Iterator end){
    return distance_helper(begin, end, Iterator::iterator_category());
}
```

tag

Traitklassen

- Lösungsansatz: Traitklassen
- Traits sind Templateklassen, die jeweils bestimmte Eigenschaften ihres Templatearguments enthalten
  - Implementiert über Templatespezialisierung
  - Definition von Eigenschaften über ein konsistentes Interface auch für primitive Typen
- Für Iteratoren: `std::iterator_traits<`*T*`>` im Header `iterator`
- Standardbibliothek enthält auch andere Traitklassen
  - `std::numeric_limits<`*T*`>` (im Header `limits`) enthält Informationen über arithmetische Datentypen
    - `std::numeric_limits<int>::max()` gibt beispielsweise die größte durch `int` darstellbare Zahl zurück (üblicherweise $2^{31} - 1$)
  - Traits zum Vergleich und zur Manipulation von Typen in `type_traits`
    - `std::is_base_of<std::ostream, std::fstream>::value` ist `true`
    - `std::remove_reference<double&>::type` ist `double`
    - Nützlich für Templatecode

Traitklassen

- Lösungsansatz: Traitklassen
- Traits sind Templateklassen, die jeweils bestimmte Eigenschaften ihres Templatearguments enthalten
  - Implementiert über Templatespezialisierung
  - Definition von Eigenschaften über ein konsistentes Interface auch für primitive Typen
- Für Iteratoren: `std::iterator_traits<T>` im Header `iterator`
- Standardbibliothek enthält auch andere Traitklassen
  - `std::numeric_limits<T>` (im Header `limits`) enthält Informationen über arithmetische Datenty
    - st darstellbare Za
  - Traits zu
    - std::is_base_of<std::ostream, std::fstream>::value ist true
    - `std::remove_reference<double&>::type` ist `double`
    - Nützlich für Templatecode

```
template<typename Iterator>
ptrdiff_t distance(Iterator begin, Iterator end){
    return distance_helper(begin, end,
         std::iterator_traits<Iterator>::iterator_category());
}
```

std::string

- `std::string` ist in gewisser Hinsicht auch ein Container
  - Enthält Elemente vom Typ `char`
- Ist als allgemein gehaltener Container implementiert:
  `std::string` ist `typedef` für `std::basic_string<char, std::char_traits<char>>`
  - Es existieren auch `wstring`, `u16string` und `u32string`
  - `char_traits` definieren Operationen auf dem Chartyp, z.B. Vergleiche
- Zugrundeliegende Implementierung entspricht der von `std::vector`
  - Ebenfalls als überalloziertes dynamisches Array implementiert
  - Besitzt die selben Memberoperationen wie `std::vector`
- Unterschied zu `vector`: `basic_string` besitzt zusätzliche stringspezifische Operationen
  - Umwandlung von/zu C-Style Strings, Vergleichsoperationen
  - `find`-Methoden zum Auffinden von Teilstrings

std::string_view (C++17)

- Referenziert Teil eines extern verwalteten Strings
- Enthält nur Start- und End-Position
- Verwendung von `std::string_view` wie `std::string`
- Beispiel:

```
std::string s = "Interesting string";
std::string_view sv = s;
std::string_view part = sv.substr(12, 6);
std::cout << part << std::endl;
```

- Ausgabe: `string`
- Achtung: Extern verwalteter String muss noch existieren!