IMS
Institut für Mikroelektronische Systeme
Leibniz Universität Hannover

Leibniz Universität Hannover

Programmierpraktikum Technische Informatik (C++)

23.6.2022

IMS
Institut für Mikroelektronische Systeme
Leibniz Universität Hannover

Leibniz
Universität
Hannover

Die Standardbibliothek
Algorithmen

Algorithmen der Standardbibliothek

- Algorithmen sind neben Containern und IO ein wichtiger Bestandteil der Standardbibliothek
- C++ bietet Algorithmen für grundlegende Operationen
  - Suchen
  - Sortieren
  - Entfernen von Elementen
  - Kopieren/Füllen
  - Aggregation (Summieren, Zählen, Minimum, ...)
  - Permutationen
- Fast alle Algorithmen liegen im Header `algorithm`
  - Einige wenige in `numeric`

Konzeption der Algorithmen

- Algorithmen könnten als Argument jeweils einen Container erhalten
  - Nachteil: Weniger flexibel, kann nicht auf Teilbereichen arbeiten
- Algorithmen erhalten als erste Argumente in der Regel zwei Iteratoren
  - Start und Ende des zu betrachtenden Bereichs
  - Halboffenes Intervall, der zweite Iterator liegt außerhalb des betrachteten Gebiets
  - Verwendung dadurch leider etwas umständlicher
- Vielen Algorithmen (Akkumulieren, ...) kann die auszuführende Operation mit übergeben werden
- Wie übergibt man Funktionen als Parameter?

Funktionspointer

- C++ unterstützt auch Pointer auf Funktionen
- Syntax: *`Returntype`*`(*`*`Name`*`)(`*`Argtypes`*`)`
  - `bool (*predicate)(int, int);` definiert eine Variable `predicate` als Pointer auf eine Funktion mit Signatur `bool(int, int)`
- `&foo` gibt einen Pointer auf die Funktion `foo` zurück
- Problematisch: Überladene Funktionen
  - Welcher Overload soll verwendet werden?
- Bestimmung über die erwartete Signatur
  - Bei direkter Übergabe (`std::find_if(...)`) möglicherweise Cast notw

```
bool isFive(int a) { return a == 5; }
//...
std::vector<int> x = {1,2,3,4,5,6,7,8,9};
bool (*predicate)(int) = &isFive;
auto iter = std::find_if(x.begin(), x.end(), predicate);
```

Probleme mit Funktionspointern

- Bei Übergabe als Parameter wird die erwartete Signatur des Funktionspointers durch die aufrufende Funktion festgelegt
- Außerdem problematisch, wenn zur Ausführung weitere Daten benötigt werden
  - Nur über globale Variablen oder lokale statische Variablen lösbar
- Aufrufe über Funktionspointer müssen zur Laufzeit aufgelöst werden und können nicht geinlined werden
- Lösungsansatz: Funktoren

Funktoren

- Zur Erinnerung: In C++ kann auch der Funktionsoperator "`operator()`" überladen werden
- Eine Klasse mit überladenem Funktionsoperator wird als Funktor bezeichnet
- Eine **Instanz** eines Funktors kann wie eine normale Funktion verwendet werden
  - Membervariablen können für persistente Daten verwendet werden
  - Erlaubt Inlining in templatisiertem Code

```
struct isVal
{
    int val;
    isVal(int v): val(v) {}
    bool operator()(const int& a) const
    { return a == this->val; }
};
//...
std::vector<int> x = {1,2,3,4,5,6,7,8,9};
auto iter = std::find_if(x.begin(), x.end(), isVal(5));
```

⟵ Wenn möglich `const` verwenden!

Lambdafunktionen (anonyme Funktionen)

- Funktoren lösen viele der Probleme von Funktionspointern
- Erzeugen aber auch neue Probleme:
  - Hoher syntaktischer Overhead für die Erstellung des Funktors
    - Funktionsoperator von Funktoren häufig nur eine Zeile, Klassendefinition deutlich länger
  - Räumliche Trennung von Funktor und dem Code, der diesen verwendet
    - Inlinedefinition der Operation direkt im verwendenden Code wäre schöner
- Lösung: Lambdafunktionen (anonyme Funktionen)
  - In den letzten Jahren immer häufiger auch in nichtfunktionalen Sprachen zu finden
  - Wesentliche (vlt. sogar wesentlichste) Neuerung in C++11

Lambdafunktionen II

- Lambdafunktionen sind Funktionsobjekte, die direkt im verwendenden Code definiert werden
- Syntax: `[](`*`Args`*`) -> `*`Returntype`*`{ `*`Body`*` }`

```
std::vector<int> x = {1,2,3,4,5,6,7,8,9};
auto iter = find_if(x.begin(), x.end(), [](int a)   -> bool
{ return a == 5; });
```

- Erzeugt eine anonyme Funktorklasse, deren Funktionsoperator den entsprechenden Code ausführt
- *`Args`* sind ganz normale Funktionsargumente
- Lambdafunktionen haben keinen (sichtbaren) Typnamen
  - Speicherung in Variablen nur über `auto`:

```
auto predicate = [](const int& a)-> bool
{ return a == 5; }
```

Trailing Return

- Definition des Rückgabetyps von Lambdas mit `-> ReturnType`
  - Abweichung von bekannter Syntax für Funktionsignaturen
- Sogenanntes Trailing Return, möglich seit C++11
- Auch für normale Funktionen möglich:

```
auto intLess(int a, int b)-> bool;
```

  - Bei normalen Funktionen wird Trailing Return durch `auto` an der eigentlichen Position des Rückgabetyps signalisiert

Impliziter Rückgabetyp

- Viele Lambdafunktionen bestehen aus einem einzigen Return-Statement
  - Explizite Angabe des Rückgabetyps scheint redundant
- Angabe des Rückgabetyps ist für Lambdas optional
- Rückgabetyp wird aus vorhandenen Return-Statements bestimmt
  - Keine Return-Statements: Rückgabetyp `void`
  - Sonst: Rückgabetyp wird aus dem Return-Statement nach den Regeln für `auto` bestimmt
  - Seit C++14: Mehreren Return-Statements erlaubt, müssen alle auf denselben Typ deduzieren

```
auto iter = std::find_if(x.begin(), x.end(), [](int a)
{ return a == 5; });
```

Impliziter Rückgabetyp für normale Funktionen

- Seit C++14 kann auch für normale Funktionen der Rückgabetyp automatisch bestimmt werden
- Signalisiert durch einen `auto`-Returntype ohne Trailing-Return
  - `auto add(int x, double y){ return x + y; }`
  - Regeln wie für Lambdas
  - Praktisch, um lange Returntypen zu vermeiden
- Weglassen des Returntypes hat aber auch Nachteile:
  - Keine Forwarddeklarationen für Funktionen mit Rückgabetyp `auto` möglich
  - Verringert Lesbarkeit, da Rückgabetyp nicht direkt ersichtlich
- Nur verwenden, wenn die folgenden Bedingungen erfüllt sind:
  - Die Funktion ist kurz genug für eine Deklaration als Inline
  - Der Rückgabetyp ist sehr lang und aus dem Quelltext sofort ersichtlich

Closures

- Bisher haben Lambdas keine Möglichkeit, Informationen neben ihren Parametern zu erhalten
  - Genau das gleiche Problem wie mit Funktionspointern
- Closures erlauben einer Lambdafunktion, auf Variablen des sie enthaltenden Scopes zuzugreifen
  - Insbesondere lokalen Variablen der Funktion, in der die Lambdafunktion definiert ist
  - Implementierung: Kopie der Variable wird in einen Datenmember des Funktors kopiert
- Verwendete Variablen müssen in sog. Capture Clause (`[]`) des Lambdas aufgeführt werden
  - Syntax: `[`*Varnames*`](`*Args*`){`*Body*`}`
  - Angabe als kommaseparierte Liste von Variablen

```
int val = 5;
auto iter = std::find_if(x.begin(), x.end(),
    [val](int a) { return a == val; });
```

Closures II

- Variablen in Capture-Clause werden bei der Erzeugung in den Funktor hineinkopiert
  - Häufig unerwünschter Performanceoverhead
  - Bei Änderungen des Wertes durch die Lambdafunktion existiert keine Möglichkeit, den Wert zurückzuerhalten
- C++ erlaubt daher auch Capture-by-Referenz
  - Variable wird mit &*Varname* in der Captureliste angegeben
  - Verhält sich in Lambda wie eine Referenz
  - **Achtung:** Sicherstellen, dass die referenzierte Variable zum Aufrufzeitpunkt noch lebt

```
int count = 0;
std::find_if(x.begin(), x.end(),
    [val, &count](int a)
    {
        if(a == 7)  ++count;
        return a == val;
    });
```

Implementation von Lambdas

```
int val = 5;
int count = 0;
std::find_if(x.begin(), x.end(), [val, &count](int a) -> bool {
    if(a == 7)  ++count;
    return a == val;
});
```

```
class main$lambda {
private:
    int  val;
    int& count;
public:
    main$lambda(int v, int& c): val(v), count(c) {}
    bool operator()(int a) const  {
        if(a == 7)  ++count;
        return a == val;
    }
};

int val = 5;
int count = 0;
std::find_if(x.begin(), x.end(), main$lambda(val, count));
```

Default-Captures

- Manuelle Angabe der Captures erlaubt präzise Kontrolle, ist aber recht mühsam
- C++ erlaubt das Angeben eines Default-Captures
  - Syntax: = für Capture-by-Value und & für Capture-by-Reference
- Alle in Lambdafunktion verwendeten Variablen, die nicht anderweitig gecaptured werden, verwenden das Defaultcapture
  - Bei Setzen eines Default-Captures dürfen keine expliziten Captureangaben den selben Modus verwenden

```
int count = 0;
std::find_if(x.begin(), x.end(),
    [=,&count](int a)
    {
        if(a == 7)  ++count;
        return a == val;
    });
```

Capture Expressions

- Bisher können Variablen nur als Kopie oder als Referenz gecaptured werden
- Problematisch für Typen wie `std::unique_ptr`, die nur Move, aber keine Kopie unterstützen
- Seit C++14: Capture Expressions
  - Syntax: *varname* = *expr*
  - Erzeugt eine Variable *varname* in Lambdafunktion, die einen automatisch bestimmten Typ hat und mit *expr* initialisiert wird
  - &*varname* = *expr* ist auch möglich, *varname* ist dann eine Referenz

Beispiel Capture Expressions

```
std::string str = "Hello!";
auto foo = [&s1=str, s2=std::move(str)]
           (const std::string& s3)
    {
        std::cout<<s1<<","<<s2<<","<<s3<<std::endl;
    };
foo(str);
```

- Annahme, dass Strings, aus denen gemoved wurde, leer sind
- Was ist die Ausgabe des Codes?
- Antwort: `",Hello!,"`
- Begründung: `s1` und `s3` sind Referenzen auf `str`, der bei Erstellung von `s2` geleert wurde

Mutable Lambdas

- Der Funktionsoperator eines Lambdas ist standardmäßig `const`
- Variablen mit Capture-by-Value können daher nicht modifiziert werden
  - Keine Auswirkungen bei Variablen mit Capture-by-Reference
- Kann mit `mutable` umgangen werden
  - `mutable` erlaubt in C++ allgemein eine Umgehung von `const`:
    `mutable`-Member dürfen auch in `const`-Objekten geändert werden
  - Für Lambdas bedeutet `mutable`, dass der Funktionsoperator nicht `const` ist

```
int count = 0;
std::find_if(x.begin(), x.end(), [=](int a) -> bool mutable
{
     if(a == 7)  ++count;
     return a == val;
});
```

Polymorphic Lambdas

- In manchen Fällen ist es wünschenswert, Lambdas mit variablen Argumenttypen zu haben
- Neuerung in C++14: Polymorphe Lambdas
- C++14 erlaubt `auto` als Typ für Argumente von Lambdafunktionen
  - Definition von Referenzparametern mit `auto&` oder `const auto&`
  - `[](const auto& a, const auto& b){ return a > b; }`
  - Erzeugt einen Funktor, dessen `operator()` ein Template ist

```
class main$lambda {
public:
    main$lambda() {}
    template<typename T, typename U>
    auto operator()(const T& a, const T& b)
    {
        return a > b;
    };
};
```

std::function

- C++ unterstützt verschiedene Arten von Funktionstypen
- Außer der Aufrufform keine Gemeinsamkeit zwischen den Arten
  - Insbesondere kein gemeinsamer Basistyp
- Welchen Parametertyp für Funktionalparameter wählen?
- Möglicher Ansatz: Templatefunktion schreiben und Typ nicht näher spezifizieren
  - Optimale Performance, da für jeden Typ explizit optimierbar
  - In der Standardbibliothek verwendeter Ansatz
  - Aber: Nicht immer möglich
- `std::function<`*`Signatur`*`>` (Header: `functional`) kann jede kompatible Art von Funktionspointer/Objekt aufnehmen
  - Kompatibel bedeutet, dass Argument- und Rückgabewerttypen nach impliziten Konvertierungen denen der `std::function` entsprechen
  - Signatur ist *`ReturnType`*`(`*`Argumenttypes`*`)`

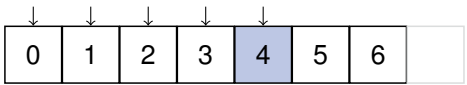
std::function II

```
void foo(std::function<void(std::stringstream&, int)> f);
void bar(std::ostream& os, int count);
//...
foo(&bar);
foo([](std::iostream& io, int count){...});
```

- Funktoren werden häufig per-Value übergeben (z.B. in der Standardbibliothek)
- `std::function` hat einen nicht zu vernachlässigenden Kopieroverhead
- Verhindert wie Funktionspointer alle Optimierungen durch Inlining
- Abwägen, ob der Templateansatz für die jeweilige Situation geeigneter wäre

Find

- Eine häufiges Problem ist das Auffinden eines bestimmten Elementes in einem unsortierten Container
- Zu diesem Zweck existiert `std::find`
  - Argumente: Start- und Enditerator sowie Wert des zu suchenden Elements
  - Rückgabe: Iterator auf das Element oder Enditerator, falls Wert nicht gefunden
  - Beispiel: `auto iter = std::find(x.begin(), x.end(), 4);`
- Lineare Suche durch den Container, Laufzeitkomplexität: $O(n)$



Find II

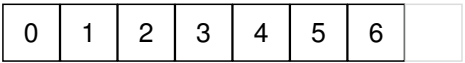
- Von Find existieren auch Varianten, die ein Prädikat statt eines Elements erhalten: `std::find_if` und `std::find_if_not`

```
auto iter = std::find_if(x.begin(), x.end(),
     [](int a) { return a == 4; });
```

- `find_if` findet das erste Element, das die Bedingung erfüllt
- `find_if_not` findet das erste Element, das die Bedingung nicht erfüllt

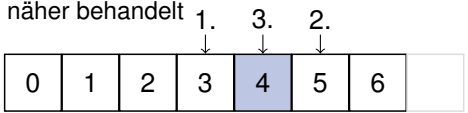
Sort

- Sortieren von Daten ist weitere häufige Aufgabe in Programmen
- Könnte über Umweg über ein `std::set` gelöst werden
  - Daten in neues `set` einfügen und sortierte Daten wieder auslesen
  - Komplizierter Ansatz mit hohem Overhead
- `std::sort(`*`begin`*`, `*`end`*`)` sortiert die Daten im angegebenen Bereich
  - *`begin`* und *`end`* **müssen** Random-Access-Iteratoren sein
  - Sortiert wird nach dem $<$-Operator
- Optimales (allgemeines) Sortierverfahren mit Laufzeitkomplexität $O(N \cdot \log(N))$
- `std::is_sorted(`*`begin`*`, `*`end`*`)` überprüft, ob der übergebene Bereich sortiert ist



Suchen in sortierten sequentiellen Containern

- `std::find` hat eine Laufzeitkomplexität von $O(n)$
- In sortierten Containern ist eine binäre Suche in $O(\log(n))$ möglich
  - Vorgehen: Überprüfen, ob das mittlere Element größer oder kleiner dem Gesuchten ist und nur die überbleibende Hälfte des Bereichs untersuchen
  - `std::find` hat keine Möglichkeit, die Sortiertheit zu überprüfen und kann sie sich daher nicht zunutze machen
- Standardbibliothek enthält Algorithmen zum Suchen in sortierten sequentiellen Containern: `std::binary_search`, `std::lower_bound`, `std::upper_bound` und `std::equal_range`
  - `std::binary_search` überprüft lediglich, ob ein bestimmtes Element existiert, daher selten sinnvoll und hier nicht näher behandelt



lower_bound, upper_bound und equal_range

- `std::lower_bound(`*begin*`,` *end*`,` *elem*`)` gibt einen Iterator auf das erste Element zurück, das **nicht kleiner** als *elem* ist
  - Achtung: Nicht zwingend Iterator auf ein Element mit Wert *elem*
  - Korrekte Position, um *elem* in den sortierten Container einzufügen
- `std::upper_bound(`*begin*`,` *end*`,` *elem*`)` gibt einen Iterator auf das erste Element, das **größer** als *elem* ist, zurück
  - Wenn *elem* nicht im Bereich, identisch mit `lower_bound`
  - Ansonsten gibt `[lower_bound, upper_bound)` den Bereich aller zu *elem* äquivalenten Elemente an
- `std::equal_range(`*begin*`,` *end*`,` *elem*`)` ist funktional identisch zu
  `std::make_pair(std::lower_bound(`*begin*`,` *end*`,` *elem*`),`
  `std::upper_bound(`*begin*`,` *end*`,` *elem*`))`
- **Wichtig:** Verhalten auf nichtsortierten Containern ist undefiniert

Vergleichsoperationen

- Für Sortierung und binäre Suchen wird normalerweise der $<$-Operator verwendet
- Manchmal nicht das gewünschte Verhalten
  - Möglicherweise ist für den Typ kein $<$-Operator definiert
  - Eventuell ist andere Anordnung gewünscht
- Algorithmen, die sortierte Daten bearbeiten, besitzen zwei Varianten
  - Die bereits bekannte Version
  - Eine Version mit einem zusätzlichen Argument *predicate*
    - `std::sort(`*begin*`, `*end*`, `*predicate*`)`

Vergleichsoperationen II

- *predicate* ist Funktionsobjekt mit Signatur `bool(T, T)`
- Muss sich wie ein $<$-Operator verhalten, also die folgenden Eigenschaften haben:
  - Transitiv: `predicate(a, b) && predicate(b, c)` $\Longrightarrow$ `predicate(a, c)` $\forall$ `a, b, c`
  - Irreflexibel: `predicate(a, a) == false` $\forall$ `a`
  - Asymmetrisch: `predicate(a, b) != predicate(b, a)` $\forall$ `a, b`
- Beispiel:
```
std::lower_bound(vec.begin(), vec.end(), 4,
[](const int& a, const int& b){ return a > b; }
```
- **Achtung:** Algorithmen, die sortierte Daten voraussetzen, müssen die zur Sortierung verwendete Ordnung verwenden
  - Das Beispiel funktioniert nicht, wenn `vec` mit `std::sort(vec.begin(), vec.end())` sortiert wurde

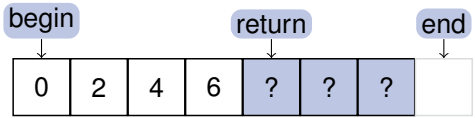
Remove

- Manuelles Entfernen aller Elemente, die bestimmte Bedingungen erfüllen, ist umständlich
  - Fehleranfällig wegen Iteratorinvalidierung
  - Für `vector` und `deque` quadratische Laufzeitkomplexität
- Lösungen in der Standardbibliothek: `std::remove(`*`begin`*`, `*`end`*`, `*`elem`*`)` und `std::remove_if(`*`begin`*`, `*`end`*`, `*`predicate`*`)`
  - Weitere Varianten für spezielle Fälle, z.B. `std::unique` entfernt alle aufeinander folgenden Duplikate
- `std::remove_if` entfernt alle Elemente, die die Bedingung erfüllen
- **Problem:** `remove_if` werden lediglich Iteratoren und nicht der Container selbst übergeben
  - Kann die Größe des Containers nicht direkt ändern

Das Remove-Erase-Idiom

- Effekte von `remove_if` und ähnlichen Algorithmen:
  - Nach Ausführung stehen die nicht gelöschten Elemente am Anfang des Bereichs
  - Rückgabewert zeigt hinter das Ende des bestehenden Bereiches
  - Rest des Containers ist in undefiniertem Zustand
- Restbereich des Containers kann mit der Bereichsform von Erase entfernt werden
  - Sogenanntes Remove-Erase-Idiom
  - Insgesamt Laufzeit: $O(n)$

```
auto pred = [](int a) { return a % 2 == 1; };
vec.erase(std::remove_if(vec.begin(), vec.end(), pred),
    vec.end());
```



Copy and Fill

- Aus C bekannt: `memcpy` und `memset`
- In C++ auch vorhanden, Verwendung aber nicht empfohlen
  - `memcpy` nur für Objekte ohne komplexe Kopieroperationen
  - `memset` ebenso problematisch, da Konstruktoren umgangen werden
  - Fehleranfällig, da Größe des Speichers in Byte angegeben
- C++ Alternativen: `std::copy` und `std::fill`

Copy and Fill II

- `std::copy(`*`srcBegin`*`, `*`srcEnd`*`, `*`dstBegin`*`)` kopiert alle Elemente aus `[`*`srcBegin`*`, `*`srcEnd`*`)` in den bei *`dstBegin`* startenden Bereich
  - Rückgabewert ist End-Iterator für den überkopierten Bereich
- `std::fill(`*`dstBegin`*`, `*`dstEnd`*`, `*`value`*`)` überschreibt alle Elemente in `[`*`dstBegin`*`, `*`dstEnd`*`)` mit *`value`*
- Varianten `std::copy_n` und `std::fill_n` erhalten einen Startiterator und eine Anzahl Elemente statt eines Bereichs
- **Achtung:** Sicherstellen, dass Zielbereich ausreichend groß für alle Elemente

Accumulate

- Häufige Aufgabe: Akkumulieren aller Elemente eines Containers
  - Beispiele: Aufsummieren, Maximum bilden, ...
- `std::accumulate(`*begin*`, `*end*`, `*accum*`, `*func*`)`
  - Definiert in `numeric`
  - *accum* gibt den Startwert für die Akkumulation an
  - *func* gibt an, wie die Werte kombiniert werden sollen
    - Rückgabetyp und erstes Argument kompatibel zum Typ von *accum*
    - Das zweite Argument kompatibel zum Typ von `*`*begin*
    - Optional, Default ist Summation
- Varianten für gängige Spezialfälle: `std::count_if`, `std::max_element`, ...

```
std::vector<unsigned> v{1,2,3,4,5,6,7,8,9,10};
double val = std::accumulate(v.begin(), v.end(), 0.0,
    [](double accum, unsigned elem)
    { return accum + 1.0 / static_cast<double>(elem); });
```

Parallelität von Algorithmen

- seit C++17: Parallele Implementierungen von Algorithmen der Standardbibliothek
- Auswahl durch Angabe von sogenannter Execution-Policy als erstes Argument
- Policies:
  - `std::seq`: Sequentielle Ausführung (single threaded execution)
  - `std::par`: Parallele Ausführung
  - `std::par_unseq`: Erlaubt weitreichendere parallele Ausführung aber gefährlich bei unzureichendem Kenntnisstand
- Beispiel:
```
std::sort(std::par, vec.begin(), vec.end(),
[](const auto& a, const auto& b){ return a.id < b.id; });
```
- Warnung: Schreibender Zugriff auf geteilte Daten kann Race-Condition (undefined behaviour) ergeben
- Empfehlung: Nur mit Lambdas verwenden, die keine geteilten Daten modifizieren

Weitere Algorithmen

- Neben den benannten Algorithmen stellt die Standardbibliothek viele weitere zur Verfügung
- Eine vollständige Auflistung finden Sie unter
  http://en.cppreference.com/w/cpp/algorithm

Algorithmen oder Memberfunktionen

- Für einige Container existiert die Funktionalität einiger Algorithmen auch als Memberfunktion
  - Beispiel: `std::list` hat die Member `.sort(...)` und `.remove(...)`
  - `std::map` besitzt ein Member `.lower_bound(...)`
- Welche Variante ist zu bevorzugen, Member oder Algorithmus?
  - Ist es möglicherweise egal?
- Erinnerung: Funktionalität ist nur als Memberfunktion implementiert, wenn dies Vorteile bietet

Algorithmen oder Memberfunktionen II

- Manchmal ist der Algorithmus nicht für den Container verwendbar
  - `map` bietet keine Random-Access-Iteratoren, `std::lower_bound` funktioniert daher nicht
- Manchmal funktioniert der generische Algorithmus, eine Implementation als Memberfunktion ist aber effektiver
  - `std::remove` verschiebt Elemente, `list.remove` verändert lediglich die Pointer

Wenn Algorithmus und Memberfunktion mit gleicher Funktionalität existieren, immer die Memberfunktion bevorzugen!

Ranges Library (C++20)

- Ranges erlauben einfachere Iteration über Teilbereiche
- Erst seit C++20 Standard, vorher ähnlich in Boost-Library
- Ranges abstrahieren
  - **Container** sind selbst auch als Ranges verwendbar
  - **Views** definieren Teilbereiche eines Containers
- Ranges können an Stelle von Containern bei Range-based-For verwendet werden
- `#include<ranges>` nötig

Views

- Namespace: `std::views`
- Views sind *„lazy“*: Filterung wird erst bei Verwendung ausgewertet
- Views können mit |-Operator auf Ranges angewendet werden und lassen sich kaskadieren
- Beispiel:

```
const std::vector numbers = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

auto even = [](int i) { return i%2 == 0; };

namespace sv = std::views;
for(const auto& i: numbers | sv::filter(even) | sv::drop(1) | sv::reverse)
    std::cout << i << ' ';
```

- Ausgabe: `10 8 6 4`, Vector `numbers` bleibt unverändert.

Algorithmen mit Ranges

- `#include<algorithm>` enthält Constrained Algorithms in Namespace `std::ranges`
- Erspart u.a. `.begin()` und `.end()` für ganzen Container
- Beispiel:

```
const std::vector v = {4, 1, 3, 2};

auto even = [](int i) { return 0 == i % 2; };

auto result = std::ranges::find_if(v, even);
if (result != v.end()) {
  std::cout << "First even element in v: " << *result << std::endl;
} else {
  std::cout << "No even elements in v" << std::endl;
}
```

- Siehe https://en.cppreference.com/w/cpp/algorithm/ranges

IMS
Institut für Mikroelektronische Systeme
Leibniz Universität Hannover

Leibniz
Universität
Hannover

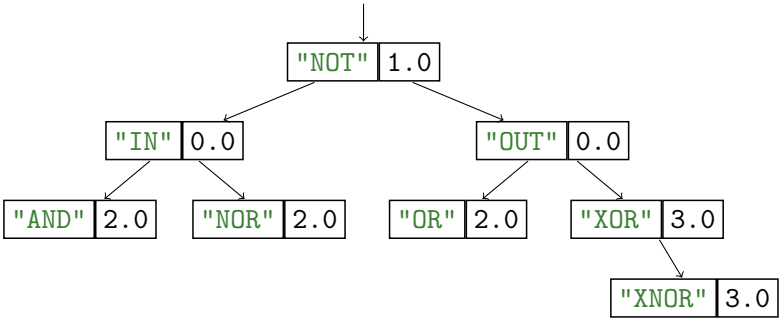
Die Standardbibliothek
Assoziative Container

Assoziative Container

- `std::map` ist ein assoziativer Container
- In assoziativen Containern hängt die Positionierung von Elementen primär vom Wert des eingefügten Elementes ab
- Wesentliche Eigenschaft: Effizientes Suchen nach Elementen
- Unterteilen sich in zwei wesentliche Kategorien:
  Sortierte und Unsortierte Container
  - Sortierte Container garantieren, dass die Elemente bei einer Iteration über den Container in bestimmter Reihenfolge durchlaufen werden
  - Unsortierte Container geben keine Garantien über die Reihenfolge
- In diesen Kategorien jeweils Unterteilung nach zwei Eigenschaften
  - Multiples Einfügen des selben Elements erlaubt oder nicht
  - Elemente sind in Schlüssel und Wert unterteilt oder nicht

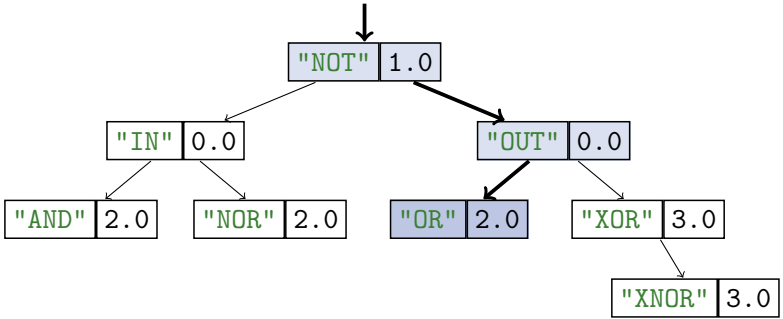
std::map

- `std::map` ist ein sortierter Container, der Elemente nach Schlüssel und Wert unterteilt und kein multiples Einfügen unterstützt
- Intern über einen balancierten binären Suchbaum abgebildet



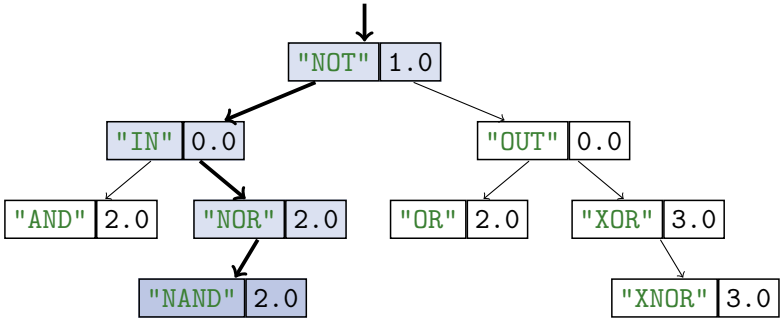
std::map

- Suchen eines bestimmten Elements (`map.find`, `map.lower_bound`, ...) ist $O(log(n))$



std::map

- Einfügen und Löschen von Elementen ebenfalls $O(log(n))$
  - Auffinden der Einfügeposition ist $O(log(n))$, einfügen selber $O(1)$
  - Varianten, die einen Iterator für die Position erhalten sind $O(1)$



Elementtyp einer Map

- `std::map<K, V>` enthält Elemente vom Typ `std::pair<const K, V>`
- `std::pair` bildet ein Wertepaar ab
  - Ähnlich zu `std::tuple`
  - Zugriff auf einzelne Werte über `std::get<N>(...)` möglich
  - `std::pair` unterstützt auch direkten Zugriff auf die Datenmember: `p.first` und `p.second`
- Iteratoren zeigen jeweils auf ein Key-Value-Paar
- Wichtig: der *Key*-Teil eines Mapelements ist `const`
  - Notwendig, da Änderungen die interne Struktur der Map korrumpieren könnten

Zugriff auf einzelne Elemente

- Mit `m[`*Key*`]` oder `m.at(`*Key*`)` kann direkt auf den einem Schlüssel zugeordneten Wert zugegriffen werden
- `m.find(`*Key*`)` gibt einen Iterator auf das jeweilige Element zurück
- Unterschied zwischen den Methoden im Verhalten bei Nichtauffinden des Schlüssels
  - `m.find(...)` gibt den End-Iterator `m.end()` zurück
  - `m.at(...)` wirft eine Exception
  - `m[...]` fügt ein neues Element mit dem entsprechenden Schlüssel und einem **defaultkonstruierten** Wert ein
- Verhalten von `m[...]` kann leicht zu Problemen führen
  - Eventuell werden Programmfehler maskiert
  - Kann nicht auf einer `const` Map ausgeführt werden
- Je nach Anwendungsfall `m.find(...)` oder `m.at(...)` verwenden

std::set

- `std::set<`*T*`>` ist konzeptionell eine `std::map` ohne Werte
  - `std::map` modelliert mit seinen Key-Value-Paaren ein Wörterbuch
  - `std::set` modelliert eine (mathematische) Menge
    - Jeder Wert darf nur einmal vorkommen und ist effizient auffindbar
- Elemente des `std::set` sind vom Typ `const` *T*
  - Aus den gleichen Gründen, aus denen Schlüssel einer Map `const` sind
- Verwendung und interne Struktur identisch zu `std::map`
  - Mit der offensichtlichen Ausnahme, dass Elemente direkt die Schlüssel sind und nicht Schlüssel-Wert-Paare

Anordnung der Elemente in sortierten Containern

- Normalerweise werden Elemente in sortierten Containern anhand des Operators $<$ angeordnet
  - Für Sets werden die Elemente verglichen, für Maps lediglich die Schlüssel
- Manchmal ist dies nicht das gewünschte Verhalten
  - Möglicherweise ist für den Typ kein $<$-Operator definiert
  - Eventuell ist andere Anordnung gewünscht
- Alle sortierten Container haben ein zusätzliches Templateargument, um dieses Verhalten zu kontrollieren:
  - `std::map<`*`Key`*`, `*`Value`*`, `*`Comp`*`>`, `std::set<`*`Elem`*`, `*`Comp`*`>`, ...
- *`Comp`* muss wie beim Sortieralgorithmus ein Funktor sein, der bzgl. Signatur und Verhalten einen $<$-Operator nachbildet

Multiset und Multimap

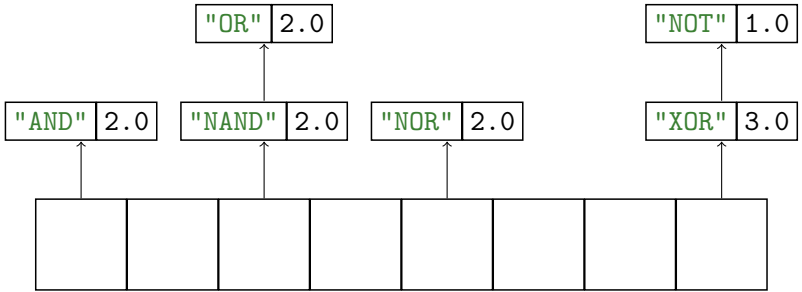
- `std::map` und `std::set` können einem Schlüssel nur jeweils einen einzigen Wert zuweisen
  - Wiederholtes `.insert` mit demselben Schlüssel führt keine Änderungen durch
  - Erfolg/Misserfolg von `.insert` über Rückgabewert ersichtlich
- Für Situation, wo Mehrfacheinfügen notwendig ist, existieren `std::multimap` und `std::multiset`
  - Verhalten und interne Struktur wie `std::map` und `std::set`, unterstützen aber mehrfaches Einfügen mit demselben Schlüssel
- In der Praxis selten notwendig, daher keine detailliertere Betrachtung in dieser Veranstaltung

Unsortierte Container

- Häufig bei assoziativen Containern nur der effiziente Elementzugriff wichtig
  - Garantien zur Reihenfolge der Elemente unnötig
  - Unnötige Garantien können Performance kosten
- Mit C++11 existiert eine neue Gruppe assoziativer Container im Standard: Unsortierte Container
  - Elementzugriff, Einfügen, Löschen in der Regel effizienter als bei sortierten Containern
  - Reihenfolge der Elemente nicht vorgegeben
    - Iteration über den Container sieht Elemente in scheinbar zufälliger Reihenfolge

std::unordered_map

- Schnittstelle von `std::unordered_map` entspricht der von `std::map`
  - Unterschiede: Elemente unsortiert und statt Vergleichsfunktion wird eine Hashfunktion verwendet
- Interne Implementierung über Hashtable

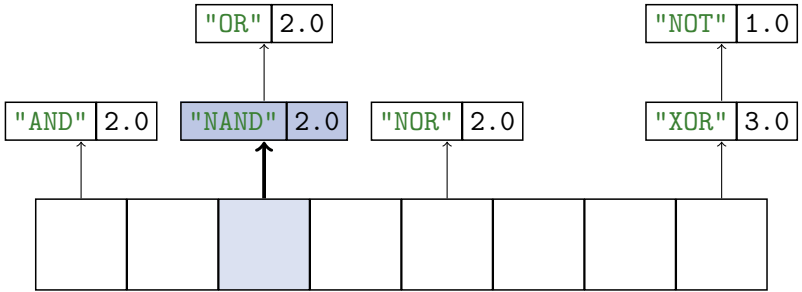


Hashtable

- Hashfunktion erzeugt für ein (beliebig komplexes) Objekt eine verkürzte Darstellung (Hash)
  - Achtung: Hashfunktionen sind nicht injektiv, derselbe Hash kann verschiedenen Objekten zugeordnet werden (Hashkollision)
  - Wichtige Eigenschaft für Hashfunktionen: Objekte, die als gleich angesehen werden, erhalten den selben Hash
  - Zweite wichtige Eigenschaft: Kollisionen möglichst selten
    - Ähnliche Objekte sollten unterschiedliche Hashwerte erhalten
- Hashtable ordnet Elementen eine ihrem Hashwert entsprechende Position in dynamischem Array zu
  - Aufgrund der Möglichkeit von Kollisionen kann jeder Arrayeintrag mehrere Objekte aufnehmen
    - Häufig enthält jeder Eintrag eine verlinkte Liste
  - Treten zuviele Kollisionen auf, wird das Array vergrößert und längerer Hashwert verwendet

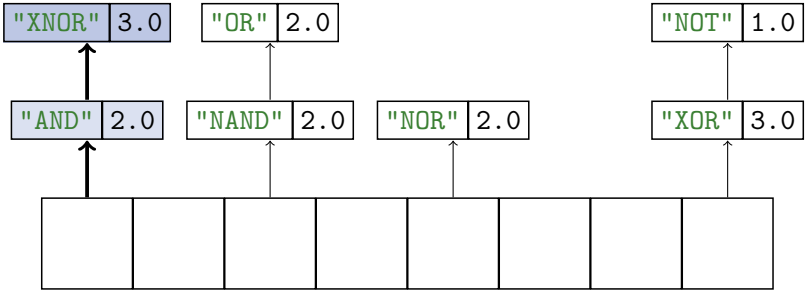
std::unordered_map

- Auffinden eines Elementes ist durchschnittlich $O(1)$
  - Bei hoher Kollisionsrate theoretisch $O(n)$
  - Tritt in der Praxis nur für extra dafür angelegte Datensätze oder schlechte Hashfunktionen auf



std::unordered_map

- Einfügen in normalen Situationen amortisiert $O(1)$
  - Normalerweise $O(1)$, außer wenn zuviele Kollisionen auftreten
  - Bei hoher Kollisionszahl vergrößern des Arrays, Rehashing aller Elemente, also $O(n)$
  - Wachstumsstrategie ähnlich zu `std::vector`



Weitere Unsortierte Container

- Wie für sortierte Container existieren von unsortierten Containern mehrere Varianten:
  - `std::unordered_map`
  - `std::unordered_set`
  - `std::unordered_multimap`
  - `std::unordered_multiset`
- Verhalten analog zu dem entsprechenden geordneten Container
- Analog zu sortierten Containern kann auch das Verhalten von unsortierten Containern über zusätzliche Templateparameter festgelegt werden
  - Statt einer Ordnungsfunktion wird eine Hashfunktion und eine Vergleichsfunktion übergeben
  - Vergleichsfunktion modelliert hier nicht $<$, sondern $==$