Programmieren in C++

Standardbibliothek (STL)

Inhalt

- Standardbibliothek in der Übersicht
- Zeiteinheiten
- Hilfsfunktionen und Hilfsklassen
- Container
- Iteratoren
- Algorithmen
- Exceptions

C++ Standardbibliothek (1)

Bibliothek	C++-Headers
Algorithms	<algorithm> <execution>
Atomic Operations	<atomic>
C Compatibility	<cassert> <cctype> <cerrno> <cfenv> <cfloat> <cinttypes> <climits> <clocale> <cmath> <csetjmp> <csignal> <cstdarg> <cstddef> <cstdint> <cstdio> <cstdlib> <cstring> <ctime> <cuchar> <cwchar> <cwctype>
Concepts	<concepts>
Containers	<array> <deque> <forward_list> <list> <map> <queue> <set> <stack> <unordered_map> <unordered_set> <vector>
Coroutines	<coroutine>
Filesystem	<filesystem>
Input/Output	<fstream> <iomanip> <ios> <iosfwd> <iostream> <istream> <ostream> <sstream> <streambuf> <syncstream>
Iterators	<iterator>

C++ Standardbibliothek (2)

Bibliothek	C++-Headers
Localization	<locale>
Numerics	<bit> <complex> <numbers> <numeric> <random> <ratio> <valarray>
Ranges	<ranges>
Regular Expressions	<regex>
Strings	<charconv> <format> <string> <string_view>
Thread Support	<barrier> <condition_variable> <future> <latch> <mutex> <semaphore> <shared_mutex> <stop_token> <thread>
Utilities	<any> <bitset> <chrono> <functional> <initializer_list> <optional> <tuple> <typeinfo> <type_traits> <utility> <variant> <memory> <memory_resource> <new> <scoped_allocator> <limits> <exception> <stdexcept> <system_error>

C++ Standard Library Headers

Wertebereiche <limits>

- Abfrage der fundamentalen Eigenschaften der numerischen Datentypen

```
numeric_limits<float>::epsilon();
```

- Range-Checker

```
template<typename T, class Pred>
bool rangeChecker(const Pred& p) {
    constexpr T min = numeric_limits<T>::min();
    constexpr T max = numeric_limits<T>::max();

    for (T i = min; i < max; i++) if (!p(i)) return false;
    if (!p(max)) return false; // verhindert Endlosschleife
    return true;
}
int main() {
    auto b = rangeChecker<uint16_t>([] (uint16_t v) {
        return v << 1 == v*2;
    });
    cout << boolalpha << b << endl;
}
```

Zeiteinheiten <chrono>

- Unterschiedliche Zeitquellen
  - system_clock
  - steady_clock
  - high_resolution_clock
- Vordefinierte Zeiteinheiten
  - Clock::time_point
  - Clock::duration
- Beispiele

```
using Clock = chrono::system_clock;
Clock::time_point start = Clock::now();
Clock::duration d = Clock::now() - start;
int64_t ns = std::chrono::nanoseconds(d).count();
using ms_t = std::chrono::duration<double, std::milli>; // new duration type
double ms = std::chrono::duration_cast<ms_t>(d).count();
```

Pair und Tuple

- Klasse pair<T1, T2> aus <utility>
  - Paarbildung von zwei wählbaren Typen
  - Einsatz: z.B. die Elemente in einer Map

```
pair<int, double> id;
int i = id.first; double d = id.second;
id = make_pair(3, 5.5);
```

- Klasse tuple aus <tuple>
  - Verallgemeinerung von pair auf beliebige Anzahl Werte beliebigen Typs

```
tuple<int, double, string> tup(1, 2.2, "drei");
auto val = get<0>(tup);
get<1>(tup) = 1.5;
size_t s = tuple_size<decltype(tup)>::value; // Anzahl Elemente im Tupel
tup = make_tuple(2, 3.3, "vier");
```

Optional und Any

- Klasse optional<T> aus <optional>
  - Objekt, welches optional einen Wert des Typs T enthält
  - gespeicherter Wert liegt innerhalb des Speicherbereichs des Objektes
  - kann nach bool gecastet werden: false, wenn kein Wert vorhanden
- Klasse any aus <any>
  - enthält Wert eines beliebigen Typs oder keinen Wert
  - kann durch Wert eines anderen beliebigen Typs überschrieben werden
- Beispiel

```
optional<int> o1;
auto o2 = make_optional<int>(10);
cout << boolalpha << o1.has_value() << endl;
cout << boolalpha << static_cast<bool>(o2) << " " << o2.value() << endl;
```

Container (1)

- Bitvektoren fixer Länge
  - bitset: <bitset>
- Halbdynamische Container
  - vector und vector<bool>: <vector>
- Listen
  - double ended queue: <deque>
  - list (doubly-linked): <list>
  - forward_list (singly-linked): <forward_list>
- Geordnete Mengen: <set>
  - set (die Schlüssel werden sortiert verwaltet)
  - multiset (Mehrfacheinträge sind erlaubt)
- Geordnete Maps: <map>
  - map
  - multimap (Schlüssel müssen nicht eindeutig sein)

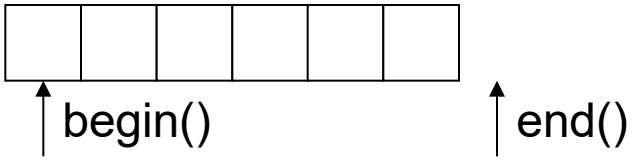
Container (2)

- Ungeordnete Mengen: <unordered_set>
  - unordered_set (die Schlüssel werden unsortiert verwaltet):
    unordered_multiset (Mehrfacheinträge sind erlaubt)
- Ungeordnete Maps: <unordered_map>
  - unordered_map
  - unordered_multimap (Schlüssel müssen nicht eindeutig sein)
- Container-Interfaces
  - verwendet einen Container (z.B., vector, deque oder list) als Datenbehälter
  - bietet spezielle Datenzugriffe an
  - Interfaces
    - stack (LIFO): <stack>
    - queue (FIFO): <queue>
    - priority queue: <queue>

Container: Datentypen und Methoden

- Datentypen (angeboten/erforderlich) für Container X<T>
  - X::value_type — Container-Element, entspricht T
  - X::reference — Referenz auf Container-Element
  - X::const_reference — dito, aber nur lesend verwendbar
  - X::iterator — Iterator
  - X::const_iterator — dito, aber nur lesend verwendbar
  - X::difference_type — vorzeichenbehafteter integraler Typ
  - X::size_type — vorzeichenloser integraler Typ für Grössenangaben
- Methoden (nicht vollständig)
  - Standard-, Kopier- und Verschiebekonstruktor, Destruktor
  - Iteratoren (lesend und schreibend): begin() und end()
  - Iteratoren (nur lesend): cbegin() und cend()
  - Grössenangaben: max_size(), size(), empty()
  - Zuweisungsoperator und Verschiebezuweisungsoperator
  - Relationale Operatoren
  - Vertauschen: swap(X&)

Iteratoren



- Konzept
  - Iterator: verallgemeinerter Zeiger, welcher auf ein Element des Containers zeigt
  - begin() und cbegin() liefern einen Zeiger, der aufs erste Element zeigt
  - end() und cend() liefern einen Zeiger, auf ein fiktives Element unmittelbar nachfolgend dem letzten Element
  - Inkrementieren ++ springt zum nächsten Element
  - Dereferenzieren * ermöglicht Zugriff aufs Element

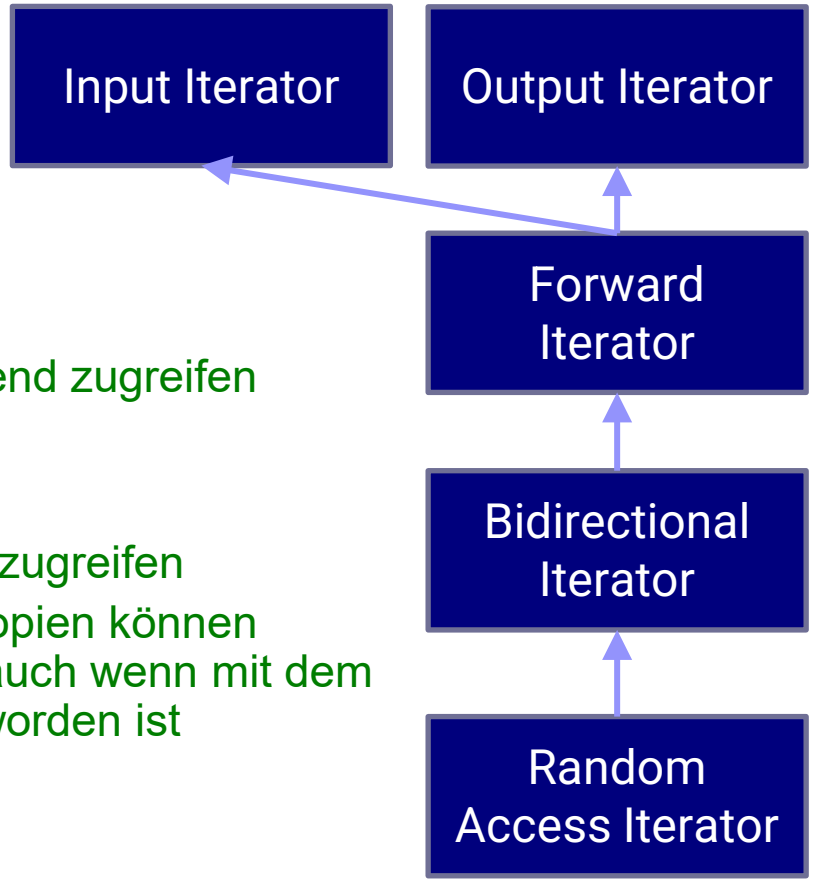
- Beispiel

```
template<class Iter> void print(Iter it, Iter end) {
    while(it != end) {
       cout << *it++ << ' ';
    }
    cout << endl;
}
```

```
vector<int> v(10);
for(size_t i = 0; i < v.size(); i++) {
     v[i] =  i;
}
print(v.cbegin(), v.cend());
```

Iterator-Kategorieren

- Input
  - nur auf das aktuelle Element lesend zugreifen
  - nur in eine Richtung iterieren
- Output
  - nur auf das aktuelle Element schreibend zugreifen
  - nur in eine Richtung iterieren
- Forward
  - lesend und schreibend auf Elemente zugreifen
  - Iterator kann mehrfach kopiert und Kopien können mehrfach wiederverwendet werden, auch wenn mit dem ursprünglichen Iterator weiteriteriert worden ist
- Bidirectional
  - kann in beide Richtungen iterieren
- Random Access
  - ermöglicht direkten, wahlfreien Zugriff (auch mit Index-Operator)



Iterator-Operationen

Operation	Input	Output	Forward	Bidirectional	Random Access
=	•		•	•	•
==	•		•	•	•
!=	•		•	•	•
*	1)	2)	•	•	•
->	•		•	•	•
++	•	•	•	•	•
--				•	•
[ ]					3)
arithmetisch					4)
relational					5)

1) Dereferenzierung ist nur lesend möglich.
2) Dereferenzierung ist nur auf der linken Seite einer Zuweisung möglich.
3) `I[n]` bedeutet `*(I+n)` für einen Iterator `I`
4) `+ += - -=` in Analogie zur Zeigerarithmetik
5) `< > <= >=` relationale Operatoren

Verschiedene Spezialiteratoren

- Move-Iteratoren
  - die Daten werden verschoben anstatt kopiert
- Insert-Iteratoren
  - mit einem Forward-Iterator kann auf Elemente lesend/schreibend zugegriffen werden
  - ein Insert-Iterator erlaubt das Einfügen in einen Container
    - front_insert_iterator
    - back_insert_iterator; Beispiel: back_insert_iterator<list<double>> bIt(…);
    - insert_iterator (einfügen an spezifizierter Position)
- Reverse-Iteratoren
  - läuft rückwärts von rbegin() bis rend()
  - ++-Operator wird zum Iterieren verwendet
- Stream-Iteratoren

Algorithmen

- Grundsätze
  - alle im Header <algorithm> vorhandenen Algorithmen sind unabhängig von einer konkreten Container-Implementierung
  - enthält eine Container-Implementierung einen gleichnamigen Algorithmus wie im Header <algorithm>, so soll die spezielle Version des Containers verwendet werden (höhere Effizienz)
  - die Algorithmen greifen über Iteratoren auf die Elemente des Containers zu
  - wird ein First- und ein End-Iterator verlangt, so ist damit das halboffene Intervall [First, End) gemeint
- Beispiel

```
const int searchValue = 5;
vector<int> v = { 9, 3, 5, 8, 1, 7, 2, 4 };

sort(v.begin(), v.end());
// get iterator to first element >= searchValue
auto pos = lower_bound(v.cbegin(), v.cend(), searchValue);
cout << *pos << endl;
```

Algorithmen: Übersicht (1)

- Suchen eines Elementes
  - find, find_if, find_end, find_first_of, adjacent_find
  - nth_element: platziert das n-te Element einer Sortierreihenfolge an die richtige Position im Array (z.B. um den Median zu bestimmen)
- Suchen einer Sequenz
  - search, search_n
- Zählen von Elementen, die ein Prädikat erfüllen
  - count
- Vergleichen zweier Elemente
  - min, max, min_element, max_element
- Vergleichen zweier Sequenzen
  - lexicographical_compare
- Vergleichen zweier Container
  - mismatch, equal

Algorithmen: Übersicht (2)

- Kopieren der Elemente eine Quellbereichs in einen Zielbereich
  - copy, copy_backward
- Vertauschen von Elementen oder Containern
  - swap, iter_swap, swap_ranges
- Einfüllen von Sequenzen
  - fill, fill_n, generate, generate_n
- Ersetzen von Elementen
  - replace, replace_if, replace_copy, replace_copy_if
- Entfernen
  - remove, remove_if, remove_copy, remove_copy_if
  - unique, unique_copy
- Transformieren (Kopieren und dabei Modifizieren)
  - transform

Algorithmen: Übersicht (3)

- Reihenfolge verändern
  - reverse, reverse_copy, rotate, rotate_copy, random_shuffle
  - partition, sort, partial_sort
- Permutationen
  - prev_permutation, next_permutation
- Suchen in sortierten Sequenzen
  - binary_search, lower_bound, upper_bound
  - equal_range
- Mischen zweier sortierter Sequenzen
  - merge, inplace_merge
- Mengenoperationen auf sortierten Strukturen
  - includes, set_union, set_intersection, set_difference, set_symmetric_difference
- Heap-Algorithmen
  - pop_heap, push_heap, make_heap, sort_heap

Parallele Algorithmen (Weiterführend)

- Parallele Ausführung
  - die meisten Algorithmen erlauben eine parallele Ausführung
  - Container sind nicht thread-safe
  - Programmierer muss Race-Conditions mit geeigneten Mitteln verhindern
- Beispiel

```
int a[] = {0, 1};
std::vector<int> v;
std::for_each(std::execution::par, std::begin(a), std::end(a), [&v] (int i) {
    v.push_back(i); // Error: data race
});
```

Fehlerbehandlung

- Exceptions
  - Hierarchie wurde überarbeitet und ergänzt
- Fehlerfälle mit Fehlercode
  - Fehler, die vom System kommen, werden dem Aufrufer als numerischer Wert zurückgeliefert
  - diese Fehlernummern sind systemabhängig und erschweren die Erstellung portablen Codes
  - Exception-Klasse system_error stellt leichtgewichtige error_code Objekte zur Verfügung
    - enthält systemspezifische Fehlercodes
    - verweist auf abstrakte, portable error_condition Objekte
  - Benutzer können eigene Fehlergruppen hinzufügen

Exceptions (1)

- Werfen von Exceptions
  - Syntax: throw *ex-object;*
  - vordefinierte Exception-Typen in <exception>-Header
  - *ex-object* kann von jedem Typ sein, auch primitiver Datentyp
- Beispiel

```
try {
    throw std::runtime_error("example");
} catch(const std::runtime_error& e) {
    std::cout << "std::runtime_error: " << e.what() << std::endl;
} catch(…) {
    std::cout << "unknown exception" << std::endl;
}
```

Exceptions (2)

- Best-Practice
  - Exceptions nur by-value werfen (automatischer Speicher)
  - Exceptions als const-Referenzen auffangen
  - nur Exceptions abgeleitet von std::exception werfen
- Exception modifizieren und weiterwerfen
  - in catch-Block das Exception-Objekt modifizieren und mit throw weiterwerfen
- noexcept
  - eine Funktion kann deklarieren, dass sie niemals eine Exception werfen wird
  - dient der Performance-Optimierung
  - Move-Semantik benötigt noexcept (würde eine Move-Operation fehlschlagen, so wären sowohl das alte als auch das neue Objekt in einem invaliden Zustand)
- Konstruktoren/Destruktor
  - Konstruktoren können Fehlschlag nur über Exceptions kommunizieren
  - während des Exception-Handlings werden evtl. Destruktoren von Attributen aufgerufen
  - Destruktoren dürfen nie Exceptions werfen