Fachhochschule Nordwestschweiz
Hochschule für Technik

Programmieren in C++

Christian Lang (Lac)

15. November 2019

Die Standard-Library im Detail

Inhalt

- Header-Überblick
- Zeit-Einheiten
- Diverse Helper
- Container-Details
- Iteratoren-Details
- Allocators
- Algorithmen
- Eigene Algorithmen implementieren
- Parallele Algorithmen

Header-Überblick

Schwerpunkte	Headers
Hilfsfunktionen & -Klassen	`<utility>` `<functional>` `<any>` `<optional>` `<variant>` `<tuple>`
Container	`<queue>` `<deque>` `<list>` `<forward_list>` `<map>` `<unordered_map>` `<set>` `<unordered_set>` `<stack>` `<vector>` `<array>` `<bitset>`
Input/Output	`<iostream>` `<istream>` `<ostream>` `<fstream>` `<sstream>` `<iomanip>` `<ios>` `<iosfwd>` `<streambuf>`
Numerisches	`<complex>` `<limits>` `<numeric>` `<valarray>`
Fehlerbehandlung	`<exception>` `<stdexcept>` `<system_error>` `<cassert>`
String	`<string>` `<charconv>` `<string_view>`
Speicher / Smart-Pointer	`<memory>` `<new>`

Header-Überblick

Schwerpunkte	Headers
Iteratoren	`<iterator>`
Algorithmen	`<algorithm>`
Typinformationen	`<typeinfo>` `<type_traits>`
Nationale Zeichensätze	`<locale>`
Parallelität	`<thread>` `<future>` `<atomic>` `<mutex>`
	`<shared_mutex>` `<condition_variable>`
Reguläre Ausdrücke	`<regex>`
Rechnen mit Einheiten	`<ratio>`
Rechnen mit Zeiteinheiten	`<chrono>`
Wahrscheinlichkeitsverteilungen	`<random>`
Dateisystem	`<filesystem>`

Nicht abschliessend

Siehe: C++ Standard Library headers

Zeit-Einheiten: `std::chrono`

- Erlaubt Zugriff auf Zeitquellen
- Definiert Zeiteinheiten
  - Zeitpunkt: `std::chrono::duration`
  - Zeitdauer: `std::chrono::time_point`

```
1  using Clock = std::chrono::system_clock;
2
3  auto start = Clock::now();
4  // Berechnungen
5  auto end = Clock::now();
6
7  using std::chrono::duration_cast;
8  auto duration = duration_cast<std::chrono::microseconds>(end - start);
9  std::cout << "measured time: " << duration.count() << "us" << std::endl;
```

Typ-Bereiche: `std::numeric_limits`

- generisch Eigenschaften von Typen evaluieren
- kann für eigene Typen spezialisiert werden

```
1  template<typename T>
2  bool CheckValue(const T& value) { ... }
3
4  template<typename T>
5  bool CheckType() {
6    constexpr auto min = std::numeric_limits<T>::min();
7    constexpr auto max = std::numeric_limits<T>::max();
8    for (T i = min; i < max; ++i) {
9      if (CheckValue(i) == false) return false;
10   }
11   if (CheckValue(max) == false) return false;
12   return true;
13 }
14
15 CheckType<bool>();
16 CheckType<uint8_t>();
```

std::pair

- erlaubt Kombination heterogener Werte
- wird in `std::map` etc. als `value_type` verwendet

```
1  std::pair<int, std::string> tup(1, "second");
2
3  // Lesen vom int Wert
4  int val = tup.first;
5
6  // Schreiben auf string Wert
7  tup.second = "empty";
```

std::tuple

- generische Version von `std::pair`
- Funktioniert das meiste auch für `std::pair`

```
1  std::tuple<int, double, std::string> tup(1, 2.2, "drei");
2
3  // Lesen vom int Wert
4  int val = std::get<0>(tup);
5
6  // Schreiben auf double Wert
7  std::get<double>(tup) = 1.5;
8
9  // Anzahl Elemente im Tuple
10 size_t s = std::tuple_size<decltype(tup)>::value;
```

std::optional

- kann Wert haben oder leer sein
- Erweiterung von `std::pair<bool, T>`
- Speicher von `T` ist immer in `std:optional`

```
1  std::optional<int> GetValue() {
2    return std::make_optional<int>(42);
3  }
4
5  auto result = GetValue();
6  if (result.has_value()) {
7    std::cout << "valid result: " << result.value() << std::endl;
8  } else {
9    std::cout << "no result" << std::endl;
10 }
```

std::any

- ein Wert von beliebigem Typ
- dynamisch und typsicher
  → Exception wenn falscher `std::any_cast`

```
1  std::any a = 1;
2  std::cout << a.type().name() << ": "
3            << std::any_cast<int>(a) << std::endl;
4
5  a = 3.14;
6  std::cout << a.type().name() << ": "
7            << std::any_cast<double>(a) << std::endl;
8
9  a.reset();
10 if (a.has_value() == false) { std::cout << "no value" << std::endl; }
```

std::variant

- typsichere union
- kann leer sein
- Alloziert kein dynamisches Memory

```
1  std::variant<int, float> v;
2  v = 12;                          // enthält nun einen int
3
4  std::cout << "get<int>=" << std::get<int>(v) << std::endl;  // Lesen von int
5  std::cout << "get<0>  =" << std::get<0>(v) << std::endl;    // dito
6
7  std::get<double>(v);             // error: kein double vorhanden
8  std::get<3>(v);                  // error: invalider Index
9  std::get<float>(v);              // exception: momentan ist int gesetzt
```

Container-Details

Container erfüllen gewisse Konventionen:

- 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
- Datentypen (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`: meistens `std::size_t`

Iterator-Kategorien

Operation	Input	Output	Forward	Bidirectional	Random Access
`=`	•		•	•	•
`==`	•		•	•	•
`!=`	•		•	•	•
`*`	1)	2)	•	•	•
`->`	•		•	•	•
`++`	•	•	•	•	•
`--`				•	•
`[]`					3)
`+ += - -=`					•
`< > <= >=`					•

1) Dereferenzierung ist nur lesend möglich.
2) Dereferenzierung ist nur auf der linken Seite der Zuweisung möglich.
3) `I[n]` bedeutet `*(I+n)` für einen Iterator `I`

Verschiedene Spezialiteratoren

- Move-Iteratoren
  - die Daten werden verschoben anstatt kopiert
  - Helper: `std::make_move_iterator`
- Insert-Iteratoren
  - Normale Iterator greifen Inplace auf Elemente zu
  - ein Insert-Iterator erlaubt das Einfügen in einen Container
    - `front_insert_iterator`
    - `back_insert_iterator`; Beispiel: `back_insert_iterator<list<double>> bIt(l);`
    - `insert_iterator` (einfügen an spezifischer Position)
- Reverse-Iteratoren
  - läuft rückwärts von `rbegin()` bis `rend()`
  - `operator++` wird zum Iterieren verwendet
  - Helper: `std::make_reverse_iterator`
- Stream-Iteratoren

Allocators

- Container müssen Speicher allozieren können
- Kontrollierbar mittels Policy: `Allocator`

```
1  // Definition von std::vector
2  template<class T, class Allocator = std::allocator<T>>
3  class vector;
4  // verwendet by default den default Allocator: std::allocator
5
6  // Manuelle Auswahl:
7  std::vector<uint32_t, custom_allocator<uint32_t>> data;
```

- Allocator ist nur für Speicher-Allozierung zuständig
- Instanziierung mittels Placement `new`

Algorithmen

- alle Algorithmen in `<algorithm>` sind unabhängig von einer konkreten Container-Implementierung
- es sollen, wenn vorhanden, die speziellen Algorithmen eines Containers verwendet werden, z.B: `std::list::sort`
- die Algorithmen greifen über Iteratoren auf die Elemente des Containers zu
- wird ein First- und ein Last-Iterator verlangt, ist damit das halboffene Intervall [begin, end) gemeint

```
1  std::vector v = { 23, 24, 25, 26, 27 };
2  auto pos = std::find_if(v.cbegin(), v.cend(), [](auto a){
3    return a == 25;
4  });
5  std::cout << std::distance(v.cbegin(), pos) << std::endl;   // 2
```

Algorithmen Übersicht

- Suchen eines Elementes
  - `find`, `find_if`, `find_end`, `find_first_of`, `adjacent_find`
- platziert das n-te Element einer Sortierreihenfolge an die richtige Position im Array (z.B. um den Median zu bestimmen)
  - `nth_element`
- 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`
- Kopieren der Elemente eines Quellbereichs in einen Zielbereich
  - `copy`, `copy_if`, `copy_backward`

Algorithmen Übersicht

- 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`
- Reihenfolge verändern
  - `reverse`, `reverse_copy`, `rotate`, `rotate_copy`, `random_shuffle`
  - `partition`, `sort`, `partial_sort`

Algorithmen Übersicht

- 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`

Nicht abschliessend

Siehe: Standard library header algorithm

Eigene Algorithmen implementieren

- mittels Iteratoren
- Iteratoren als Templates
- Iterator-Intervall [begin, end) einhalten

```
1  template<typename Iterator, typename T>
2  Iterator Find(Iterator begin, Iterator end, const T& value) {
3    for (auto it = begin; it != end; ++it) {
4      if (*it == value) {
5        return it;
6      }
7    }
8    return end;
9  }
10
11 std::vector<int> data = {1, 3, 6, 7, 8};
12 auto it = Find(data.cbegin(), data.cend(), 6);
```

Parallele Algorithmen

- die meisten Algorithmen erlauben eine parallelisierte Ausführung
- steuern mit Execution Policy
  - `std::execution::sequenced_policy`
  - `std::execution::parallel_policy`
  - `std::execution::parallel_unsequenced_policy`
  - `std::execution::unsequenced_policy`
- Pogrammierer ist für Race-Conditions verantwortlich

```
1  int a[] = {0, 1};
2  std::vector<int> v;
3  std::for_each(std::execution::par, std::begin(a), std::end(a), [&v](int i) {
4    v.push_back(i);   // Error: data race
5  });
```