Fachhochschule Nordwestschweiz
Hochschule für Technik

Programmieren in C++

Christian Lang (Lac)

27. September 2019

Einführung in die Standard-Library

Inhalt

- Überblick
- Entstehung
- Container
- Smart-Pointer
- Grundkonzept Templates
- Iteratoren

Überblick

- Ausnahmebehandlung (Exceptions)
- Strings
- Container
- (numerische) Algorithmen
- Iteratoren
- Spezielle Datentypen: komplexe Zahlen, Zeit, Tuple
- Threads und Synchronisationsprimitive
- Speicherverwaltung
- Ein- und Ausgabe
- reguläre Ausdrücke
- Zufallszahlen und Wahrscheinlichkeitsverteilungen
- Type-Traits
- Helper für alles mögliche (z.B: Meta-Programming)
- etc.

Entstehung

- Standard Library von C (`libc`)
  - entwickelt während Sprach-Standardisierung (1983-1989)
  - POSIX spezifiziert
- Standard Template Library (STL)
  - entwickelt von Alexander Stepanov et al. (1979-1994) bei HP
  - basiert auf SGI's STL, ist aber nicht identisch
  - Polymorphismus zur Kompilationszeit (templates)
- Standard Library von C++ (`libstdc++`)
  - Umfasst `libc`, STL und zusätzliche hilfreiche Klassen und Funktionen
  - für C++11 komplett überarbeitet und erweitert worden

Nomenklatur

Heute spricht man im C++-Umfeld nur noch von der Standard-Library, meint damit aber auch immer alle Teile aus der STL und der `libc`.

Container (Auszug)

- `std::array`
- `std::vector`
- `std::string`
- `std::stack`
- `std::bitset`
- `std::list`, `std::forward_list`
- `std::set`, `std::unordered_set`
- `std::map`, `std::unordered_map`
- `std::multiset`, `std::unordered_multiset`
- `std::multimap`, `std::unordered_multimap`
- `std::deque`, `std::priority_queue`
- `std::initializer_list`
- etc.

Smart-Pointer

Reference-Counting and strong Ownership

- `std::shared_ptr`
- `std::weak_ptr`
- `std::unique_ptr`

Deprecated

- `std::auto_ptr`

Grundkonzept Templates

- Generische Klassen und Funktionen in C++
- Template = Schablone = parametrisierbarer Typ
- Templates: typsicher | Makros: nur Textersetzung
- Quellcode vereinfachen und Länge reduzieren

```
1  // Klassen-Template
2  template<typename T>
3  class vector {
4    ...
5  };
6
7  std::vector<int> data;
```

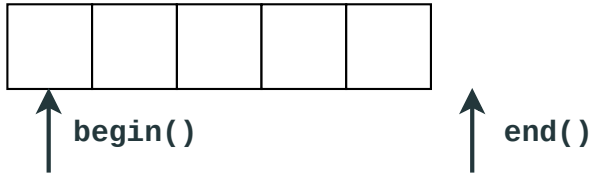
```
1  // Funktions-Template
2  template<typename T>
3  T min(T a, T b) {
4    return (a < b) ? a : b;
5  }
6
7  int r = std::min<int>(3, 4);
```

Template-Fakten

- erhöht die Compile-Zeit
- keine Source-Files mehr → nur noch Header
- In `template<...>` kann `class` oder `typename` verwendet werden
- Best-Practice:
  - `class` für alle Typen ohne primitive
  - `typename` für alle Typen

Iteratoren

- Iteratoren sind verallgemeinertes Konzept für Pointer auf Container
- Container bieten `std::begin()` und `std::end()`
- `std::cbegin()` und `std::cend()` für unveränderbare Iteratoren
- Iteratoren erlauben Operatoren: `++` und `*`



Beispiel: Iteratoren

```
1  template<class T>
2  void PrintAll(const T& c) {
3    T::const_iterator it = c.cbegin(), end = c.cend();
4
5    while(it != end) {
6      std::cout << *it++ << ' ';
7    }
8  }
```

Untypisch

In der Standard-Library wird jeweils nicht der Container selber übergeben, sondern nur die Iteratoren `begin` und `end`.

Iteratoren vs. Pointer

- Überall wo Iteratoren erlaubt sind können auch Pointer verwendet werden
- Freie Funktionen `std::begin()` und `std::end()` funktionieren für C++-Container und C-Arrays.
- Range-based Loop basiert auf `std::begin()` und `std::end()`

```
1  template<class T>
2  void PrintAll(const T& c) {
3    for(auto it = std::begin(c); it != std::end(c); ++it) {
4      std::cout << *it << ' ';
5    }
6  }
7
8  template<class T>
9  void PrintAll(const T& c) {
10   for(const auto& val : c) {     // range based loop
11     std::cout << val << ' ';
12   }
13 }
```