Container, Iteratoren und Algorithmen

☑ **Lösung zu Aufgabe 10.1** Die erhaltenen Ausgaben sind wie folgt (inklusive Kommentar mit der betroffenen Funktion zur einfacheren Übersicht):

```
1 2 3 4 5 6 7 8 9 // std::vector
9 8 7 6 5 4 3 2 1 // std::reverse
9 3 7 6 5 4 3 2 1 // std::replace
9 3 7 3 3 4 3 2 1 // std::fill
9 3 7 3 3 4 3 7 3 // std::copy
9 7 7 4 3 3 3 3 3 // std::sort
9 4 3 3 3 3 3 3 3 // std::remove
```

☑ **Lösung zu Aufgabe 10.2** Wir erklären in der folgenden Auflistung das jeweilige Problem und die zugehörige Lösung für alle der fünf betroffenen Programmzeilen:

- `std::list<int>`: Dieser Behälter ist üblicherweise als doppelt verkettete Liste mit beidseitigen Zeigern zwischen je zwei aufeinanderfolgenden Einträgen implementiert. Die Iteratoren dieses Behälters unterstützen daher keinen *Random Access*, was aber von der Funktion `std::sort` vorausgesetzt wird. Statt dieser generischen Sortierfunktion kann man aber auch eine (zwar weniger effiziente) klasseneigene Methode verwenden, die mit `l.sort()` aufgerufen werden kann.
- `std::vector<int>`: Es ist sicher möglich, einen Vektor (ein Array) mit einem vorgegebenen Wert zu befüllen. Jedoch werden im gezeigten Code aus Versehen *Const-Iteratoren* (beispielsweise mit der Syntax `v.cbegin()` anstatt `c.begin()`) an die Funktion übergeben. Über diese Art von Iteratoren darf aber (ähnlich wie bei Const-Referenzen) der Behälter nicht verändert werden.
- `std::forward_list<int>`: Hierbei handelt es sich wieder um eine verkettete Liste, aber jetzt gehen die Zeiger nur in die Vorwärts-Richtung. Die zugehörigen Iteratoren sind nur von der Kategorie *Forward* aber nicht *Bidirectional*, was aber in der Implementation der Funktion `std::reverse` vorausgesetzt wird. Die Lösung ist hier wieder, die klasseneigene Methode `f.reverse()` aufzurufen.
- `std::set<Person>`: Dieser Behälter ist in der Standard-Bibliothek als balancierter Suchbaum implementiert. Die Implementation eines solchen Suchbaums verlangt immer danach, die enthaltenen Werte anhand ihrer Grösse zu vergleichen. Die Lösung ist also, für unsere eigene Klasse `Person` einen Operator `bool operator<(const Person&, const Person&)` zu implementieren.
- `std::unordered_set<Person>`: Dies ist eine sogenannte Hash-Tabelle. Sie verwendet intern den Vergleichsoperator `bool operator==(const Person&, const Person&)`, den wir deshalb zuerst definieren müssen. Des weiteren muss eine Hash-Funktion vorhanden sein, die wir mittels Spezialisierung des Klassen-Templates `std::hash<Person>` zur Verfügung stellen können.

☑ **Lösung zu Aufgabe 10.3** Es ist möglich, die Funktion wie folgt mit *Forward-Iteratoren* zu implementieren. Weitere Erklärungen sind in den Kommentaren von folgendem Code zu finden:

```
template <typename Iterator> // Ein Typ-Parameter fuer Iterator reicht
void update_max_element(
    Iterator begin, Iterator end, typename Iterator::value_type x) {
  if(begin != end) { // Stelle sicher, dass Eingabe nicht leer ist
    typename Iterator::value_type max = *begin;
    for(Iterator it = begin; it != end; ++it) {
      if(*it > max) max = *it; // Durchlauf 1: Finde das Maximum
    }
    for(Iterator it = begin; it != end; ++it) {
      if(*it == max) *it = x; // Durchlauf 2: Schreibe neue Werte
    }
  }
}
```