30 Algorithmen

Dieses Kapitel behandelt die folgenden Themen:

- Zusammenarbeit mit Iteratoren und Containern
- Algorithmen mit Prädikat
- Algorithmen mit binärem Prädikat
- Übersicht

Alle im Header `<algorithm>` vorhandenen Algorithmen sind unabhängig von der speziellen Implementierung der Container, auf denen sie arbeiten. Sie kennen nur Iteratoren, über die auf die Datenstrukturen in Containern zugegriffen werden kann. Die Iteratoren müssen nur wenigen Kriterien genügen (siehe Kapitel 11.2). Dadurch bedingt können Iteratoren sowohl komplexe Objekte als auch einfache Zeiger sein. Bei der Übergabe zweier Iteratoren gilt die übliche, auf Seite 767 beschriebene Definition für Intervalle.

Alle Algorithmen sind im Namespace `std`. Sie sind von der speziellen Implementierung der Container, auf denen sie arbeiten, vollständig getrennt. Sie kennen nur Iteratoren, über die auf die Datenstrukturen in Containern zugegriffen werden kann. Manche Algorithmen tragen denselben Namen wie Container-Methoden. Durch die Art des Gebrauchs tritt jedoch keine Verwechslung auf. Die vollständige Trennung kann aber auch Nachteile haben: Ein sehr allgemeiner Algorithmus `find()` wird einen Container gelegentlich vom Anfang bis zum Ende durchsuchen müssen. Die Komplexität ist $O(N)$, wobei N die Anzahl der Elemente des Containers ist. Bei Kenntnis der Container-Struktur könnte `find()`

sehr viel schneller sein. Zum Beispiel ist die Komplexität der Suche in einem sortierten Set-Container nur $O(\log N)$. Deshalb gibt es einige Algorithmen, die unter demselben Namen sowohl als allgemeiner Algorithmus als auch als Elementfunktion eines Containers auftreten. Normalerweise ist die maßgeschneiderte Elementfunktion vorzuziehen.

30.1 Algorithmen mit Prädikat

Mit Prädikat ist ein Funktionsobjekt gemeint, das einem Algorithmus mitgegeben wird, und das einen Wert vom Typ `bool` zurückgibt, wenn es auf einen dereferenzierten Iterator angewendet wird. Anstelle des Funktors kann es auch eine Funktion sein. Der dereferenzierte Iterator ist nichts anderes als eine Referenz auf ein Objekt, das im Container abgelegt ist. Das Funktionsobjekt soll ermitteln, ob dieses Objekt eine bestimmte Eigenschaft hat. Nur wenn diese Frage mit `true` beantwortet wird, findet der Algorithmus auf dieses Objekt Anwendung. Ein allgemeines Schema dafür ist:

```
template <class InputIterator, class Predicate>
void algorithm(InputIterator first,
               InputIterator last,
               Predicate pred) {
   while (first != last) {
       if(pred(*first)) {          // gilt Prädikat?
           show_it(*first);        // ... oder andere Funktion
       }
       ++first;
   }
}
```

Die Klasse `Predicate` darf ein Objekt nicht verändern. Einige Algorithmen, die Prädikate benutzen, haben eine Endung `_if` im Namen, andere nicht. Allen gemeinsam ist, dass ein Prädikat in der Parameterliste erwartet wird.

Ein Beispiel für die Anwendung eines unären Prädikats sehen Sie auf der Seite 661.

30.1.1 Algorithmen mit binärem Prädikat

Ein binäres Prädikat verlangt zwei Argumente. Damit kann eine Bedingung für zwei Objekte im Container formuliert werden, zum Beispiel ein Vergleich. Der Algorithmus könnte folgenden Kern enthalten:

```
if(binary_pred(*first, *second)) { // gilt Prädikat?
    do_something_with(*first, *second);
// ...
```

In diesem Sinn könnten auch Funktionsobjekte als binäres Prädikat verwendet werden. Der zweite Parameter eines binären Prädikats braucht allerdings kein Iterator zu sein:

```
template <class InputIterator,
```

```
        class binaryPredicate,
        class T>
void another_algorithm(InputIterator first,
                       InputIterator last,
                       binaryPredicate bpred,
                       T aValue) {
   while (first != last) {
       if(bpred(*first, aValue)) {
          show_it(*first);
       }
       ++first;
   }
}
```

Die Anwendung binärer Prädikate sehen Sie u. a. auf den Seiten 660 und 693.

Übungen

30.1 Erweitern Sie die Lösung zu Aufgabe 28.4, indem Sie mit dem Algorithmus `count_if` ermitteln, wie viele Personen eines bestimmten Rangs vorhanden sind.

30.2 Erweitern Sie die Lösung zu Aufgabe 28.4, indem Sie mithilfe des Algorithmus `equal_range` alle Personen eines bestimmten Rangs anzeigen.

30.2 Übersicht

Der C++-Standard unterteilt die Algorithmen in die Bereiche

- Nicht-verändernde Algorithmen. Diese Algorithmen lassen die Elemente eines Containers unverändert (z.B. suchen).
- Verändernde Algorithmen. Diese Algorithmen ändern die Elemente eines Containers (z.B. Container mit Werten füllen).
- Sortieren und verwandte Algorithmen. Dazu gehören auch die binäre Suche, Algorithmen für Mengen, Heap-Algorithmen und andere.
- Algorithmen der C-Bibliothek. Damit sind die C-Algorithmen `qsort()` (Quicksort) und `bsearch()` (binäre Suche) gemeint.
- Verallgemeinerte numerische Algorithmen.

Dieses Buch führt die Algorithmen nicht entsprechend der Aufteilung im C++-Standard auf. Ein Algorithmus ist zur Lösung eines Problems gedacht, weswegen eine problemorientierte Darstellung bevorzugt wird. Das Kapitel 24 stellt Algorithmen für verschiedene Aufgaben vor.

Wenn Sie einen Algorithmus suchen, sehen Sie am besten im Inhaltsverzeichnis bei Kapitel 24 nach, ob die zu lösende Aufgabe dort aufgeführt ist. Wenn Sie jedoch gezielt nach einem bestimmten STL-Algorithmus suchen, helfen Ihnen das Register des Buchs und die nachfolgenden Übersichtstabellen.

Tabelle 30.1: Verändernde Algorithmen

Algorithmus	Header	siehe Abschnitt	Seite
copy	<algorithm>	24.12.5	717
copy_n	<algorithm>	24.12.5	719
copy_if	<algorithm>	24.12.5	718
copy_backward	<algorithm>	24.12.5	717
fill	<algorithm>	24.3.2	648
fill_n	<algorithm>	24.3.2	648
generate	<algorithm>	24.3.3	648
generate_n	<algorithm>	24.3.3	648
iter_swap	<algorithm>	24.12.6	719
is_partitioned	<algorithm>	siehe [ISOC++, 25.3.13]	
move	<algorithm>	27.2	749
move_backward	<algorithm>	27.2	749
partition	<algorithm>	24.4.1	666
partition_copy	<algorithm>	siehe [ISOC++, 25.3.13]	
partition_point	<algorithm>	siehe [ISOC++, 25.3.13]	
random_shuffle	<algorithm>	24.3.12	657
remove	<algorithm>	24.12.9	723
remove_if	<algorithm>	24.12.9	723
remove_copy	<algorithm>	24.12.9	723
remove_copy_if	<algorithm>	24.12.9	723
replace	<algorithm>	24.12.8	722
replace_if	<algorithm>	24.12.8	722
replace_copy	<algorithm>	24.12.8	722
replace_copy_if	<algorithm>	24.12.8	722
reverse	<algorithm>	24.3.14	660
reverse_copy	<algorithm>	24.3.14	660
rotate	<algorithm>	24.3.11	656
rotate_copy	<algorithm>	24.3.11	656
stable_partition	<algorithm>	24.4.1	666
swap	<algorithm>	24.12.6	719
swap_ranges	<algorithm>	24.12.6	719
transform	<algorithm>	24.12.7	720
unique	<algorithm>	24.3.13	658
unique_copy	<algorithm>	24.3.13	658

Tabelle 30.2: Sortieren und Verwandtes

Algorithmus	Header	siehe Abschnitt	Seite
binary_search	<algorithm>	24.5.6	681
equal_range	<algorithm>	24.5.6	682
includes	<algorithm>	24.6.1	684
inplace_merge	<algorithm>	24.4.6	673
is_heap	<algorithm>	24.7.5	692
is_heap_until	<algorithm>	24.7.5	692
is_sorted	<algorithm>	siehe [ISOC++, 25.4.1.5]	
is_sorted_until	<algorithm>	siehe [ISOC++, 25.4.1.5]	
lexicographical_compare	<algorithm>	24.3.18	665
lower_bound	<algorithm>	24.5.6	682
make_heap	<algorithm>	24.7.3	691
max	<algorithm>	24.12.11	726
max_element	<algorithm>	24.3.10	655
merge	<algorithm>	24.4.6	671
min	<algorithm>	24.12.11	726
min_element	<algorithm>	24.3.10	655
minmax	<algorithm>	24.12.11	726
minmax_element	<algorithm>	24.3.10	655
next_permutation	<algorithm>	24.3.17	663
nth_element	<algorithm>	24.4.5	669
partial_sort	<algorithm>	24.4.4	669
partial_sort_copy	<algorithm>	24.4.4	669
pop_heap	<algorithm>	24.7.1	689
prev_permutation	<algorithm>	24.3.17	663
push_heap	<algorithm>	24.7.2	690
set_difference	<algorithm>	24.6.4	686
set_intersection	<algorithm>	24.6.3	686
set_symmetric_difference	<algorithm>	24.6.5	687
set_union	<algorithm>	24.6.2	685
sort	<algorithm>	24.4.2	667
sort_heap	<algorithm>	24.7.4	691
stable_sort	<algorithm>	24.4.3	667
upper_bound	<algorithm>	24.5.6	682

Tabelle 30.3: Nicht-verändernde Algorithmen

Algorithmus	Header	siehe Abschnitt	Seite
adjacent_find	<algorithm>	24.5.4	679
all_of	<algorithm>	24.3.16	662
any_of	<algorithm>	24.3.16	662
count	<algorithm>	24.3.15	661
count_if	<algorithm>	24.3.15	661
equal	<algorithm>	24.8.2	694
find	<algorithm>	24.5.1	674
find_end	<algorithm>	24.5.3	678
find_first_of	<algorithm>	24.5.2	675
for_each	<algorithm>	24.12.3	716
mismatch	<algorithm>	24.8.1	692
none_of	<algorithm>	24.3.16	662
search	<algorithm>	24.5.3	677
search_n	<algorithm>	24.5.5	680

Tabelle 30.4: Verallgemeinerte numerische Algorithmen

Algorithmus	Header	siehe Abschnitt	Seite
accumulate	<numeric>	24.3.5	650
adjacent_difference	<numeric>	24.3.9	653
inner_product	<numeric>	24.3.7	651
iota	<numeric>	24.3.4	649
partial_sum	<numeric>	24.3.8	653

Tabelle 30.5: Algorithmen der C-Library

Algorithmus	Header	siehe Abschnitt	Seite
bsearch	<cstdlib>	35.8	878
qsort	<cstdlib>	5.9	224