STEFAN KUHLINS
MARTIN SCHADER

Die C++ Standardbibliothek

4. Auflage

Einführung
und Nachschlagewerk

Springer

Die C++-Standardbibliothek

Stefan Kuhlins · Martin Schader

Die C++-Standardbibliothek

Einführung und Nachschlagewerk

Vierte, durchgesehene Auflage

Mit 77 Abbildungen und 37 Tabellen

Springer

Dr. Stefan Kuhlins
Professor Dr. Martin Schader
Universität Mannheim
Lehrstuhl für Wirtschaftsinformatik III
Schloss
68131 Mannheim

Die 1. Auflage erschien 1999 in der Reihe Objekttechnologie
ISBN 3-540-65052-0

ISBN 3-540-25693-8 4. Auflage
Springer Berlin Heidelberg New York
ISBN 3-540-43212-4 3. Auflage Springer Berlin Heidelberg New York

Bibliografische Information Der Deutschen Bibliothek
Die Deutsche Bibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über <http://dnb.ddb.de> abrufbar.

Dieses Werk ist urheberrechtlich geschützt. Die dadurch begründeten Rechte, insbesondere die der Übersetzung, des Nachdrucks, des Vortrags, der Entnahme von Abbildungen und Tabellen, der Funksendung, der Mikroverfilmung oder der Vervielfältigung auf anderen Wegen und der Speicherung in Datenverarbeitungsanlagen, bleiben, auch bei nur auszugsweiser Verwertung, vorbehalten. Eine Vervielfältigung dieses Werkes oder von Teilen dieses Werkes ist auch im Einzelfall nur in den Grenzen der gesetzlichen Bestimmungen des Urheberrechtsgesetzes der Bundesrepublik Deutschland vom 9. September 1965 in der jeweils geltenden Fassung zulässig. Sie ist grundsätzlich vergütungspflichtig. Zuwiderhandlungen unterliegen den Strafbestimmungen des Urheberrechtsgesetzes.

Springer ist ein Unternehmen von Springer Science+Business Media

springer.de

© Springer-Verlag Berlin Heidelberg 2001, 2002, 2005
Printed in Germany

Die Wiedergabe von Gebrauchsnamen, Handelsnamen, Warenbezeichnungen usw. in diesem Werk berechtigt auch ohne besondere Kennzeichnung nicht zu der Annahme, dass solche Namen im Sinne der Warenzeichen- und Markenschutz-Gesetzgebung als frei zu betrachten wären und daher von jedermann benutzt werden dürften.

SPIN 11418061 43/3153-5 4 3 2 1 0 – Gedruckt auf säurefreiem Papier

Vorwort

Warum wird eine Standardbibliothek für C++ gebraucht? Programmentwickler benötigen immer wieder die gleichen Datenstrukturen wie z. B. dynamische Felder. Aufgrund verschiedener konkreter Anforderungen werden einzelne Implementierungen aber in der Regel sehr unterschiedlich ausfallen. Dadurch werden Programme anderer Entwickler schwer verständlich, und beim Wechsel der eingesetzten Bibliotheken entsteht erheblicher Lernaufwand. Durch eine Standardbibliothek lassen sich diese Probleme in den Griff bekommen. Um sich an die verschiedenen Anforderungen, unter denen eine Bibliothek zum Einsatz kommt, anpassen zu können, ist eine wichtige Voraussetzung für den Erfolg einer Standardbibliothek, dass sie flexibel und erweiterbar ist. Mit der *Standard Template Library* (STL), dem Kernstück der aktuellen C++-Standardbibliothek, wird dieses Ziel erreicht.

Darüber hinaus ist die STL äußerst effizient, so dass sich eine hervorragende Performance erreichen lässt. Allerdings setzt ein Gewinn bringender Einsatz der STL ein tiefes Verständnis für das Design der STL voraus. Um die STL zu verstehen, genügt es nicht, lediglich eine Funktionsreferenz zu lesen. Deshalb haben wir in diesem Buch besonderen Wert darauf gelegt, die Konzepte und Funktionsweisen der STL zu erläutern sowie auf typische Stolpersteine hinzuweisen. In diesem Zusammenhang ist vor allem Kapitel 2 hervorzuheben, das sich mit der Konzeption und Entwicklung der STL beschäftigt.

Bei der Beschreibung der einzelnen Komponenten der STL – Container, Iteratoren, Algorithmen, Funktionsobjekte usw. – gehen wir auch auf Implementierungsdetails ein, die vom Standard zwar nicht festgeschrieben sind, sich in der Regel aber aufgrund von Vorgaben für die Komplexität automatisch ergeben. Das Wissen um Implementierungsdetails unterstützt eine effiziente Benutzung der STL. Ferner demonstrieren zahlreiche kleine Anwendungsbeispiele den Gebrauch.

Im Schatten der STL existieren viele weitere interessante Komponenten der C++-Standardbibliothek, zu denen unter anderem *Streams*, *Strings*, *Auto-Pointer*, *Bitsets* und komplexe Zahlen gehören. Auch für diese Klassen liefern wir die für den praktischen Einsatz notwendigen Informationen.

Unsere Ausführungen basieren auf dem aktuellen C++-Standard, der unter der Bezeichnung ISO/IEC 14882:2003, *International Standard for the C++ Programming Language* (*Second Edition*), erschienen ist. Das Dokument kann beim *American National Standards Institute* (ANSI) bezogen werden. Der Download einer elektronischen Version ist derzeit für $18 von http://www.ansi.org/ möglich. Eine ältere Version vom Dezember 1996 ist unter http://www.dkuug.dk/jtc1/sc22/open/n2356/ frei zugänglich. Vom C++-Standardkomitee unter http://www.open-

std.org/jtc1/sc22/wg21/ diskutierte Korrekturen und Verbesserungsvorschläge haben wir weitgehend berücksichtigt.

Gegenüber der Urfassung der STL, die von *Alexander Stepanov* und *Meng Lee* bei *Hewlett Packard* als Technischer Report HPL-94-34 mit dem Titel „*The Standard Template Library*“ im April 1994 publiziert wurde, sind während der Standardisierung unzählige Änderungen vorgenommen worden. An Stellen, an denen es uns nützlich erscheint, weisen wir auf frühere Besonderheiten hin, damit sich unsere Programmbeispiele an ältere Versionen der STL anpassen lassen.

Da wir hier den C++-Standard beschreiben, sollten sich alle unsere Beispielprogramme mit jedem standardkonformen C++-Compiler erfolgreich übersetzen lassen. Zum Testen unserer Programmbeispiele haben wir die folgenden C++-Compiler eingesetzt (in alphabetischer Reihenfolge):

- Borland C++ unter Windows
- GNU g++ unter Linux und Solaris sowie Windows (cygwin)
- Kuck & Associates, Inc. KAI C++ unter Linux
- Microsoft Visual C++ unter Windows

Jedes Beispielprogramm lässt sich mit mindestens einem dieser Compiler erfolgreich übersetzen. Darüber hinaus decken die im Text angegebenen Definitionen für Klassen, Funktionen usw. einen Großteil der STL ab. Insbesondere älteren Implementierungen kann man damit auf die „Sprünge“ helfen, so dass man seine Programme gemäß dem aktuellen Standard schreiben kann und späterer Umstellungsaufwand entfällt.

Im Internet existieren frei verfügbare Implementierungen der STL und interessante Erweiterungen. Derzeit erscheinen uns insbesondere *STLport* und *Boost* (http://www.stlport.org/ sowie http://www.boost.org/) erwähnenswert.

Die C++-Standardbibliothek reizt die Programmiersprache C++ voll aus. Das geht sogar so weit, dass die Sprache für die Bibliothek erweitert wurde; *Member-Templates* sind dafür ein Beispiel (siehe Seite 70). Um die in diesem Buch dargestellten Themen verstehen zu können, muss man über fundierte C++-Kenntnisse verfügen. Zum Erlernen der Sprache C++ und als Nachschlagewerk empfehlen wir gerne unser Buch „*Programmieren in C++*“, das ebenfalls im Springer-Verlag erschienen ist.

Wir gehen davon aus, dass das Buch in der vorgegebenen Reihenfolge durchgearbeitet wird. Um langweilende Wiederholungen zu vermeiden, werden z. B. Elementfunktionen von Containerklassen nur beim ersten Auftreten im Text ausführlich und mit Beispielen erläutert.

Jeweils am Kapitelende haben wir einige Aufgaben zusammengestellt. Lösungsvorschläge befinden sich in Kapitel 15. Die meisten Aufgaben beschäftigen sich mit Themen, die über das im Text Gesagte hinausgehen. Dementsprechend

ist der Schwierigkeitsgrad recht hoch, und ein Blick auf die Lösungen lohnt sich.

Für Informationen rund um das Buch haben wir im *World Wide Web* eine Homepage unter http://www.wifo.uni-mannheim.de/veroeff/stl/ eingerichtet. Dort finden Sie:

- den Sourcecode aller mit dem Diskettensymbol 🖫 gekennzeichneten Programme und Lösungen;
- sämtliche Programmfragmente – Sie brauchen also nichts abzutippen;
- aktuelle Ergänzungen und Korrekturen.

Außerdem können Sie sich dort in unsere *Mailing-Liste* eintragen, damit wir Sie mit den jeweils neuesten Informationen zum Buch auf dem Laufenden halten können.

Über Anregungen unserer Leserinnen und Leser an unsere Postanschrift oder als E-Mail an stlbuch@wifo.uni-mannheim.de würden wir uns freuen. Wer uns einen Fehler (gleichgültig welcher Art) zuerst mitteilt, den erwähnen wir namentlich im *World Wide Web* auf unseren Seiten zum Buch.

Wir danken unseren Lesern – und hier insbesondere Rüdiger Dreier, Frank Fasse, Christian Hoffmann, Wolfgang Kaisers, Arnold Ludwig, Thomas Mehring, Olaf Raeke und Peter Schatte – für hilfreiche Hinweise zu den ersten drei Auflagen. Außerdem gilt unser Dank dem Springer-Verlag für die, wie immer, sehr gute Zusammenarbeit.

Stefan Kuhlins, Martin Schader

Inhaltsverzeichnis

1 Vorbemerkungen 1

1.1 namespace std ... 1
1.2 Header-Dateien ... 2
1.3 Eigenschaften ... 2
1.4 Kanonische Klassen ... 4
1.5 Komplexität und Aufwand ... 5

2 Konzeption und Entwicklung der STL 7

2.1 Eine Feldklasse ... 7
2.2 Eine Listenklasse ... 8
2.3 Folgerungen ... 9
2.4 Standardisierung der Suchfunktion ... 9
2.5 Die Schnittstelle eines Iterators ... 11
2.6 Ein Iterator für die Feldklasse ... 12
2.7 Ein Iterator für die Listenklasse ... 13
2.8 Zusammenfassung ... 13
2.9 const-Korrektheit ... 14
2.10 Flexible Suche mit Funktionsobjekten ... 16
2.11 Kleinste Bausteine ... 18
2.12 Programmübersetzungs- und -laufzeit ... 20
2.13 Der Rückgabetyp für eine Zählfunktion ... 22
2.14 Fehlerquellen ... 25
2.15 Ist die STL objektorientiert? ... 27
2.16 Einsatz der Standardbibliothek ... 29
2.17 Aufgaben ... 30

3 Funktionsobjekte 33

3.1 Basisklassen für Funktionsobjekte ... 33
3.2 Arithmetische, logische und Vergleichsoperationen ... 35
3.3 Projektionen ... 38
3.3.1 binder1st ... 38
3.3.2 bind1st ... 39
3.3.3 binder2nd ... 39
3.3.4 bind2nd ... 40
3.3.5 Beispiel ... 40
3.4 Negativierer ... 41
3.4.1 unary_negate ... 41
3.4.2 not1 ... 42
3.4.3 binary_negate ... 42
3.4.4 not2 ... 42
3.5 Adapter für Funktionszeiger ... 43
3.5.1 pointer_to_unary_function ... 44

3.5.2 ptr_fun für einstellige Funktionen ... 44
3.5.3 pointer_to_binary_function ... 44
3.5.4 ptr_fun für zweistellige Funktionen ... 45
3.6 Adapter für Zeiger auf Elementfunktionen ... 45
3.6.1 mem_fun_ref_t und const_mem_fun_ref_t ... 47
3.6.2 mem_fun_ref für Elementfunktionen ohne Argument ... 47
3.6.3 mem_fun1_ref_t und const_mem_fun1_ref_t ... 48
3.6.4 mem_fun_ref für Elementfunktionen mit Argument ... 49
3.6.5 mem_fun_t und const_mem_fun_t ... 49
3.6.6 mem_fun für Elementfunktionen ohne Argument ... 50
3.6.7 mem_fun1_t und const_mem_fun1_t ... 50
3.6.8 mem_fun für Elementfunktionen mit Argument ... 51
3.7 Funktionen zusammensetzen ... 51
3.8 Aufgaben ... 52

4 Hilfsmittel 55

4.1 Vergleichsoperatoren ... 55
4.2 pair ... 56
4.3 Aufgaben ... 57

5 Container 59

5.1 vector ... 60
5.2 Allgemeine Anforderungen an Container ... 62
5.3 Anforderungen an reversible Container ... 68
5.4 Anforderungen an sequenzielle Container ... 68
5.5 Optionale Anforderungen an sequenzielle Container ... 73
5.6 Weitere Funktionen ... 76
5.7 deque ... 79
5.8 list ... 84
5.9 Auswahl nach Aufwand ... 90
5.10 Aufgaben ... 91

6 Containeradapter 93

6.1 stack ... 93
6.2 queue ... 95
6.3 priority_queue ... 96
6.4 Aufgaben ... 99

7 Assoziative Container 101

7.1 map ... 102
7.2 Anforderungen an assoziative Container ... 104
7.3 Der Indexoperator der Klasse map ... 109
7.4 multimap ... 110
7.5 set ... 113
7.6 multiset ... 116
7.7 Elemente in set und multiset modifizieren ... 118
7.8 Übersicht der Container ... 120
7.9 Aufgaben ... 120

8 Iteratoren 123

8.1 Iteratoranforderungen ... 123
8.2 Input-Iteratoren ... 125
8.3 Output-Iteratoren ... 126
8.4 Forward-Iteratoren ... 127
8.5 Bidirectional-Iteratoren ... 129
8.6 Random-Access-Iteratoren ... 129
8.7 Übersicht über die Iteratorkategorien ... 131
8.8 Hilfsklassen und -funktionen für Iteratoren ... 132
8.8.1 iterator_traits ... 132
8.8.2 Die Basisklasse iterator ... 134
8.8.3 Iteratorfunktionen ... 135
8.8.3.1 advance ... 135
8.8.3.2 distance ... 136
8.9 Reverse-Iteratoren ... 137
8.10 Insert-Iteratoren ... 140
8.10.1 back_insert_iterator ... 141
8.10.2 front_insert_iterator ... 142
8.10.3 insert_iterator ... 143
8.11 Stream-Iteratoren ... 144
8.11.1 istream_iterator ... 145
8.11.2 ostream_iterator ... 147
8.12 Aufgaben ... 148

9 Algorithmen 151

9.1 Übersicht ... 153
9.2 Nichtmodifizierende Algorithmen ... 157
9.2.1 for_each ... 157
9.2.2 find und find_if ... 158
9.2.3 find_end ... 160
9.2.4 find_first_of ... 161
9.2.5 adjacent_find ... 163
9.2.6 count und count_if ... 164
9.2.7 mismatch ... 165
9.2.8 equal ... 167
9.2.9 search ... 168
9.2.10 search_n ... 169
9.3 Modifizierende Algorithmen ... 170
9.3.1 copy ... 170
9.3.2 copy_backward ... 172
9.3.3 swap ... 173
9.3.4 iter_swap ... 173
9.3.5 swap_ranges ... 174
9.3.6 transform ... 175
9.3.7 replace und replace_if ... 177
9.3.8 replace_copy und replace_copy_if ... 178
9.3.9 fill und fill_n ... 179

9.3.10 generate und generate_n ... 181
9.3.11 remove_copy und remove_copy_if ... 182
9.3.12 remove und remove_if ... 183
9.3.13 unique_copy ... 185
9.3.14 unique ... 187
9.3.15 reverse ... 188
9.3.16 reverse_copy ... 189
9.3.17 rotate ... 190
9.3.18 rotate_copy ... 191
9.3.19 random_shuffle ... 192
9.3.20 partition und stable_partition ... 194
9.4 Sortieren und ähnliche Operationen ... 195
9.4.1 sort ... 195
9.4.2 stable_sort ... 197
9.4.3 partial_sort ... 198
9.4.4 partial_sort_copy ... 199
9.4.5 nth_element ... 200
9.5 Binäre Suchalgorithmen ... 202
9.5.1 lower_bound ... 202
9.5.2 upper_bound ... 203
9.5.3 equal_range ... 204
9.5.4 binary_search ... 206
9.5.5 Schlüsselsuche mit Funktionsobjekt ... 207
9.6 Mischalgorithmen ... 208
9.6.1 merge ... 208
9.6.2 inplace_merge ... 209
9.7 Mengenalgorithmen für sortierte Bereiche ... 210
9.7.1 includes ... 211
9.7.2 set_union ... 212
9.7.3 set_intersection ... 213
9.7.4 set_difference ... 215
9.7.5 set_symmetric_difference ... 216
9.7.6 Mengenalgorithmen für multiset-Objekte ... 217
9.8 Heap-Algorithmen ... 218
9.8.1 make_heap ... 219
9.8.2 pop_heap ... 219
9.8.3 push_heap ... 219
9.8.4 sort_heap ... 220
9.8.5 Beispielprogramm ... 220
9.9 Minimum und Maximum ... 221
9.9.1 min ... 221
9.9.2 max ... 222
9.9.3 min_element ... 223
9.9.4 max_element ... 224
9.10 Permutationen ... 225
9.10.1 lexicographical_compare ... 225
9.10.2 next_permutation ... 226

9.10.3 prev_permutation 227
9.11 Numerische Algorithmen 229
9.11.1 accumulate 229
9.11.2 inner_product 230
9.11.3 adjacent_difference 231
9.11.4 partial_sum 231
9.12 Erweitern der Bibliothek mit eigenen Algorithmen 232
9.13 Präfix- versus Postfixoperatoren 234
9.14 Aufgaben 235

10 Allokatoren 239

10.1 Der Standardallokator 239
10.2 allocator<void> 243
10.3 Aufgaben 243

11 Strings 245

11.1 Containereigenschaften 246
11.2 basic_string 248
11.3 Implementierungsdetails 262
11.4 Sortieren von Strings 263
11.5 Aufgaben 264

12 Streams 267

12.1 Überblick 267
12.2 ios_base 270
12.2.1 Formatierung 272
12.2.2 Streamstatus 277
12.2.3 Initialisierung 278
12.2.4 Nationale Einstellungen 278
12.2.5 Synchronisation 280
12.3 basic_ios 280
12.3.1 Statusfunktionen 282
12.3.2 Ausnahmen 283
12.4 basic_ostream 284
12.4.1 sentry 285
12.4.2 Formatierte Ausgaben 287
12.4.3 Unformatierte Ausgaben 288
12.5 basic_istream 288
12.5.1 sentry 290
12.5.2 Formatierte Eingaben 290
12.5.3 Unformatierte Eingaben 291
12.6 basic_iostream 294
12.7 Ein- und Ausgabe von Objekten benutzerdefinierter Klassen 295
12.8 Namensdeklarationen 296
12.9 Manipulatoren 297
12.9.1 Manipulatoren ohne Parameter 297
12.9.2 Manipulatoren mit einem Parameter 300
12.10 Positionieren von Streams 301

12.11 Streams für Dateien ... 303
12.11.1 Modi zum Öffnen von Dateien ... 303
12.11.2 Die Header-Datei <fstream> ... 304
12.12 Streams für Strings ... 307
12.13 Aufgaben ... 309

13 Weitere Komponenten der C++-Standardbibliothek 311

13.1 auto_ptr ... 311
13.2 bitset ... 317
13.3 vector<bool> ... 323
13.4 complex ... 325
13.5 numeric_limits ... 331
13.6 valarray ... 337
13.6.1 slice und slice_array ... 343
13.6.2 gslice und gslice_array ... 346
13.6.3 mask_array ... 348
13.6.4 indirect_array ... 350
13.7 Aufgaben ... 351

14 Zeiger in Containern verwalten 353

14.1 Beispielklassen ... 353
14.2 Ein set-Objekt verwaltet Zeiger ... 356
14.3 Smart-Pointer ... 357
14.4 Ein set-Objekt verwaltet Smart-Pointer ... 358
14.5 Ein set-Objekt verwaltet Zeiger mittels Funktionsobjekten ... 359
14.6 Ein map-Objekt verwaltet Zeiger ... 361
14.7 Aufgaben ... 363

15 Lösungen 365

Anhang 399

A Die Containerklasse slist ... 399
B Die Klasse LogAllocator ... 413
C Literatur ... 415

Index 416

1 Vorbemerkungen

Vorab möchten wir auf einige Neuerungen wie den Namensbereich std und die Benennung von Header-Dateien aufmerksam machen. Außerdem stellen wir *kanonische* Klassen und die *O-Notation* vor.

1.1 namespace std

Für die Standardbibliothek ist der Namensbereich std reserviert. Alle Komponenten der Standardbibliothek und damit auch der STL sind im namespace std definiert. Der Klassiker "Hello, world!" sieht gemäß aktuellem C++-Standard so aus:

```
#include <iostream>

int main() {
    std::cout << "Hello, world!" << std::endl;
}
```

Bei regem Gebrauch der Standardbibliothek, wovon für die meisten C++-Programme auszugehen ist, wird die Angabe des Namensbereichs std schnell lästig. Abhilfe kann man mit einer using-Direktive schaffen:

```
#include <iostream>
using namespace std;

int main() {
    cout << "Hello, world!" << endl;
}
```

Wenn wie im Beispiel die Funktion main nicht mit einer return-Anweisung beendet wird, ergänzt der Compiler automatisch return 0; zum Signalisieren eines erfolgreichen Programmablaufs.

Im weiteren Text werden wir bei Definitionen von Komponenten der Standardbibliothek den Namensbereich std nicht mehr angeben. Beispielsweise schreiben wir bei der Definition der Containerklasse vector (vgl. Seite 62) einfach

```
template<class T, class Allocator = allocator<T> >
class vector {
    // ...
};
```

statt

```
namespace std {
    template<class T, class Allocator = allocator<T> >
    class vector {
        // ...
    };
    // ...
}
```

1.2 Header-Dateien

Wie das Beispielprogramm "Hello, world!" zeigt (siehe Seite 1), entfällt die Dateiendung .h beim Einbinden von Header-Dateien der Standardbibliothek. Aus Gründen der Abwärtskompatibilität können Header-Dateien auch noch in der früheren Form mit .h eingebunden werden. Dies hat den Effekt, dass die so eingebundenen Komponenten global deklariert sind und nicht in einem Namensbereich. Man sollte davon für zukünftige Projekte keinen Gebrauch mehr machen.

Die alten Header-Dateien der C-Bibliothek, wie z. B. <stdlib.h>, werden jetzt in der Form <cstdlib> angegeben, d. h. mit einem führenden c und ohne .h. Die Deklarationen sind dann (mit Ausnahme von Makros) in den Namensbereich std eingebettet.

Zum Instanzieren von Template-Klassen und –Funktionen muss der Compiler auf den Sourcecode zugreifen können. Die STL liegt daher komplett im Sourcecode vor, der über Header-Dateien eingebunden wird, die man sich mit jedem beliebigen Editor ansehen kann. Allerdings sollte man aufpassen, dass nicht versehentlich Änderungen gespeichert werden!

1.3 Eigenschaften

Um Wiederholungen in späteren Kapiteln zu vermeiden, führen wir in diesem Abschnitt einige Begriffe für besondere Eigenschaften ein, auf die dann verwiesen wird. Beim ersten Lesen kann dieser Abschnitt übersprungen werden.

Ein Typ T heißt *copy-constructible*, wenn Objekte des Typs in der üblichen Weise mittels Copy-Konstruktor erzeugt werden können. Formal gilt mit T t; und const T u; für den Copy-Konstruktor, dass t == T(t) und u == T(u) ist. Außerdem muss der Destruktor ~T() zum Zerstören der Objekte ausführbar sein. Damit eine selbst entwickelte Klasse X *copy-constructible* ist, sollten demnach der Copy-Konstruktor in der Form X(const X&) und der Destruktor ~X() definiert sein.

T ist darüber hinaus *assignable*, wenn nach Zuweisungen der Form t = u der Wert von t gleich dem von u ist. Der Rückgabetyp soll wie üblich T& sein. Für eine selbst entwickelte Klasse X ist also gegebenenfalls ein Zuweisungsoperator in der Form X& operator=(const X&) zu definieren.

Zum Sortieren von Objekten benutzt die Standardbibliothek den Operator <. Formal ausgedrückt wird ein Typ T *less-than-comparable* genannt, wenn für Objekte a und b des Typs T das Ergebnis von a < b in den Typ bool konvertierbar ist und < eine *Ordnungsrelation* definiert. Für den Operator < muss dabei gelten, dass

- a < a den Wert false liefert und
- aus a < b && b < c == true folgt, dass a < c == true ist.

Eine selbst entwickelte Klasse X kann man durch einen geeigneten überladenen Kleineroperator der Form bool operator<(const X&, const X&) oder mit einer entsprechenden Elementfunktion bool operator<(const X&) const *less-than-comparable* machen.

Eine weitere, flexiblere Möglichkeit zum Sortieren von Objekten stellen Vergleichsfunktionen beziehungsweise entsprechende Funktionsobjekte dar, die anstelle des Operators < eingesetzt werden. Wie Operator < soll auch eine Vergleichsfunktion namens comp eine Ordnungsrelation definieren. Für Objekte a, b und c des Typs T muss dann entsprechend gelten, dass

- comp(a, a) == false ist und
- aus comp(a, b) && comp(b, c) == true folgt, dass comp(a, c) == true ist.

Wenn außerdem eine Vergleichsfunktion equiv(a, b) als !comp(a, b) && !comp(b, a) definiert wird, dann verlangt man weiterhin, dass

- aus equiv(a, b) && equiv(b, c) == true folgt, dass equiv(a, c) == true ist.

Mit equiv(a, b) wird ausgedrückt, dass zwei Objekte a und b eines Typs T *äquivalent* sind. Bezogen auf den Standardfall, in dem comp dem Operator < entspricht, entspricht equiv dem Operator ==. Statt a == b kann dann auch !(a < b) && !(b < a) formuliert werden. Die Standardbibliothek benutzt derartige Ausdrücke unter anderem beim Suchen nach Objekten, wobei nur Operator < beziehungsweise eine entsprechende Vergleichsfunktion comp vorausgesetzt wird. Es wird nicht noch eine weitere Funktion für Äquivalenz beziehungsweise Gleichheit gefordert.

Ein Typ T heißt *equality-comparable*, wenn für ihn der Operator == so definiert ist, dass für Objekte a, b und c des Typs T das Ergebnis von a == b in den Typ bool konvertierbar ist und gilt, dass

- a == a den Wert true liefert,
- aus a == b folgt, dass b == a ist und
- aus a == b && b == c folgt, dass a == c ist.

Eine selbst entwickelte Klasse X kann man durch einen geeigneten überladenen Gleichheitsoperator der Form bool operator==(const X&, const X&) oder mit einer entsprechenden Elementfunktion bool operator==(const X&) const *equality-comparable* machen.

Bemerkung: Wenn für eine Vergleichsfunktion comp der Ausdruck !comp(a, b) && !comp(b, a) den Wert true liefert, heißt das im Allgemeinen nicht, dass a und b identisch sind. Zum Beispiel sind bei einer Sortierung von Zeichenketten, die keinen Unterschied zwischen großen und kleinen Buchstaben macht, die Objekte "max", "Max" und "MAX" äquivalent, obwohl sie offensichtlich nicht denselben Wert haben.

1.4 Kanonische Klassen

In C++ gibt es einige Elementfunktionen, die für die meisten Klassen und insbesondere für Klassen, die als Typ für Containerelemente vorgesehen sind, bereitgestellt werden sollten. Dazu gehören Standardkonstruktor, Destruktor, Copy-Konstruktor, Zuweisungsoperator und Vergleichsoperatoren. Eine Klasse, die diese Funktionen definiert, bezeichnen wir als *kanonische* Klasse. Sie besitzt die Eigenschaften *copy-constructible*, *assignable*, *less-than-comparable* und *equality-comparable*, z. B.

```
class X {
public:
    X();                                    // Standardkonstruktor
    ~X() throw();                           // Destruktor
    X(const X&);                            // Copy-Konstruktor
    X& operator=(const X&);                 // Zuweisungsoperator
    bool operator==(const X&) const;        // Gleichheitsoperator
    bool operator<(const X&) const;         // Kleineroperator
    // ... je nach Bedarf weitere Elemente
};
```

Falls der Compiler geeignete Versionen für Standardkonstruktor, Destruktor, Copy-Konstruktor und Zuweisungsoperator automatisch generiert, kann auf explizite Definitionen verzichtet werden.

Ein Standardkonstruktor ist im Zusammenhang mit Containerklassen nicht zwingend notwendig, sofern dies nicht explizit anders angegeben wird. Allerdings benutzen einige Funktionen den Standardkonstruktor für die Deklaration von Standardargumenten, siehe z. B. Seite 69. Wenn solche Funktionen unter Benutzung von Standardargumenten aufgerufen werden, muss für einen Typ T der Ausdruck T() definiert sein.

Ein Destruktor sollte generell keine Ausnahmen auswerfen. Anderenfalls könnte ein Programm abrupt durch die Funktion terminate beendet werden, wenn beim *Stack-unwinding*, das Folge einer ausgeworfenen Ausnahme ist, eine weitere Ausnahme ausgeworfen wird.

Die Vergleichsoperatoren können auch als globale (dann meist friend-) Funktionen definiert werden. Wenn für die Objekte einer Klasse keine Ordnung besteht, kann der Kleineroperator entfallen. Die anderen Vergleichsoperatoren (!=, >, >= und <=) sind in wieder verwendbarer Form in der Header-Datei <utility>