V.

Teil V:
Die C++-Standardbibliothek

26

Aufbau und Übersicht

Dieses Kapitel behandelt die folgenden Themen:

- Aufbau der Standardbibliothek
- Auslassungen/Anmerkungen zur Darstellung
- Bezug zu den Beispielen des Buchs

Dieser Teil des Buchs ist als *Nachschlagewerk*, nicht als Einführung für die wichtigsten Elemente der C++-Standardbibliothek gedacht, sofern sie in den vorangegangenen Kapiteln noch nicht ausreichend beschrieben werden. Ein gutes C++-Verständnis, möglicherweise erworben durch den ersten Teil, ist bei der Lektüre hilfreich. Allein der sehr dichte Text des Standarddokuments umfasst mehr als 1300 Seiten. Sämtliche technischen Details hier aufzuführen, würde ebenso wie der Versuch, alle Bestandteile der Bibliothek zu erläutern, den Rahmen dieses Buchs sprengen. Der Schwerpunkt liegt daher auf den Informationen, die für den Anwender wesentlich sind. Auf Dinge, an denen eher die Hersteller von Compilern interessiert sind, wird verzichtet. Den an weitergehenden Einzelheiten interessierten Leserinnen und Lesern sei als Ergänzung [BeckP], [KL] und [Str] empfohlen. Die Standarddokumente [ISOC] und [ISOC++] selbst kann man natürlich auch zu Rate ziehen.

Tabelle 26.1: Aufbau der Standardbibliothek (ohne C-Header)

Schwerpunkt	Header	Hinweise/Behandlung auf Seite(n)
Hilfsfunktionen und -klassen	`<utility>`	750
	`<functional>`	753
Container	`<bitset>`	801
	`<deque>`	775
	`<list>`	772
	`<map>`	782
	`<unordered_map>`	793
	`<queue>`	777
	`<set>`	787
	`<unordered_set>`	798
	`<stack>`	776
	`<vector>`	770
Iteratoren	`<iterator>`	805
Algorithmen	`<algorithm>`	815
Ein-/Ausgabe	`<fstream>`	96, 220, 390
	`<iomanip>`	383
	`<ios>`	375
	`<iostream>`	93
	`<istream>`	380, 735
	`<ostream>`	377
	`<sstream>`	393
Nationale Besonderheiten	`<locale>`	821
Numerisches	`<complex>`	695
	`<limits>`	725
	`<numerics>`	Tabelle 30.4, Seite 820
	`<valarray>`	857
String	`<string>`	841
Laufzeittyperkennung	`<typeinfo>`	295
Fehlerbehandlung	`<exception>`	301, 308
	`<stdexcept>`	301, 308
Speicher	`<memory>`	855
	`<new>`	854

Zu jedem Compiler gibt es eine Bibliothek mit nützlichen Klassen und Routinen. Die C++-Standardbibliothek (englisch *standard library*) stellt ein erweiterbares Rahmenwerk mit folgenden Komponenten zur Verfügung: Diagnose, Strings, Container, Algorithmen, komplexe Zahlen, numerische Algorithmen, Anpassung an nationale Zeichensätze, Ein-/Ausgabe und anderes mehr.

Die C++-Standardbibliothek ist systematisch nach Schwerpunkten aufgebaut, denen verschiedene Header zum Einbinden zugeordnet sind. Die Tabellen 26.1 und 26.2 geben eine Übersicht über die wichtigsten Header und über die Seitenzahl, wo sie ggf. beschrieben werden. Dabei kann es sein, dass dort kaum Informationen stehen, nämlich dann,

wenn es sich um aus der Programmiersprache C kommende Header handelt, die durch C++ überflüssig geworden sind. Einzelheiten über Auslassungen folgen. Alle Datentypen, Klassen und Funktionen sind im Namespace `std`, sodass diese Tatsache des Weiteren nicht mehr erwähnt werden muss. In den Tabellen 26.1 und 26.2 sind Doppelnennungen vermieden worden, wie zum Beispiel `<cmath>` auch bei »Numerisches« aufzuführen. Aus historischen Gründen sind manche Dinge nicht dort zu finden, wo man sie erwartet.

Tabelle 26.2: Ausgewählte C-Header der Standardbibliothek

Schwerpunkt	Header	Seite
C-Header	`<cassert>`	874
	`<cctype>`	874
	`<cerrno>`	875
	`<cmath>`	875
	`<cstdarg>`	876
	`<cstddef>`	877
	`<cstdlib>`	877
	`<cstring>`	879
	`<ctime>`	881

Zum Beispiel ist die Funktion `fabs()` (Absolutbetrag von `float`-Zahlen) sinnvollerweise im Header `<cmath>` enthalten, die Funktion `abs()` (Absolutbetrag von `int`-Zahlen) jedoch im Header `<cstdlib>`. Um Wiederholungen zu vermeiden, sind allgemeine Informationen, die für viele Abschnitte gelten, nur einmal am Anfang aufgeführt. Wenn zum Beispiel etwas über Vektoren nachgeschlagen werden soll (Abschnitt 28.2.1), ist es daher empfehlenswert, auch am Anfang des Kapitels über Container (Seite 763) nachzusehen.

26.1 Auslassungen

Einige C-Header

Einige von der Programmiersprache C herrührende Header sind in diesem Buch zwar erwähnt, aber nicht beschrieben, weil sie durch C++ überflüssig geworden sind. Dazu gehört zum Beispiel der Header `<cstdio>` für die Ein- und Ausgabe, dessen Funktionen wegen der `iostream`-Bibliothek nicht mehr notwendig sind. Ferner werden die C-Header für Wide-Character- und Multibyte-Zeichen weggelassen, zumal die Unterstützung durch die String-Klasse gewährleistet ist.

Vereinfachte Template-Schreibweise

Der einfacheren Lesbarkeit wegen werden manche Templates auf Zeichen des Typs `char` bezogen, sodass zum Beispiel die beiden Deklarationen (aus der Klasse `bitset`)

```
template<typename charT, class traits, class Allocator>
basic_string<charT, traits, Allocator> to_string() const;
```

```
template<typename charT, class traits, size_t N>
basic_ostream<charT, traits>&
operator<<(basic_ostream<charT, traits>& os, const bitset<N>& x);
```

vereinfacht werden zu

```
string to_string() const;

template <size_t N>
ostream& operator<<(ostream& os, const bitset<N>& x);
```

Die Standardbibliothek stellt einen Standard-Allokator (Klasse `allocator`) zur Speicherbeschaffung zur Verfügung. Die Speicherverwaltung mit eigenen Allokatoren ist möglich, aber relativ speziell und nur in seltenen Fällen notwendig, sodass hier stets vom Standard-Allokator ausgegangen wird. Daher werden im Folgenden nur die vereinfachten Deklarationen verwendet, wie gezeigt:

```
// vollständige Deklaration
template<typename T, class Allocator = allocator<T> > class vector;
// vereinfachte Deklaration
template<typename T> class vector;
```

Es ergeben sich keinerlei Probleme dadurch, weil der Vorgabeparameter für den Allokator stets zuletzt kommt und deswegen bei Bedarf eingefügt werden kann.

Container-Methoden für R-Wert-Referenzen

Manche Container-Methoden haben ein Gegenstück für R-Wert-Referenzen. Zum Beispiel fügt die Methode `push_front(const T& x)` der Klasse `list` das Element `x` des Typs `T` am Anfang ein, indem eine Kopie von `x` in der Liste angelegt wird. Falls ein R-Wert übergeben wird, ist eine Kopie nur Zeitverschwendung; er kann dann ohne Kopie direkt in die Liste eingetragen werden. Dafür gibt es die überladene Methode `push_front(T&& x)`. Die Bedeutung, nämlich ein Objekt am Anfang der Liste einzutragen, ist dieselbe. Aus diesem Grund werden überladene Methoden mit `T&&`-Parametertypen, obwohl vorhanden, zur Abkürzung bei den Containern in Kapitel 28 nicht mit aufgeführt.

Type Traits

In [ISOC++] gibt es über 50 Template-Klassen mit Typnamen, die zur Compilierzeit ausgewertet werden können. Diese Template-Klassen werden »traits« genannt. Auf einige davon und auf die Wirkungsweise wird in Kapitel 29 ab Seite 805 eingegangen, weswegen auf die Auflistung aller übrigen verzichtet wird. Abschnitt 29.1.1 zeigt ein Beispiel. [BeckP] und [ISOC++] bieten weitere Informationen.

forward_list

`forward_list` ist eine einfach verkettete Liste. Weil die doppelt verkettete Liste `list` ebensogut oder besser einsetzbar ist, wird auf eine nähere Beschreibung verzichtet.

26.2 Beispiele des Buchs und die C++-Standardbibliothek

Der Sinn dieses Buchs ist es, nicht nur als Nachschlagewerk zu dienen, sondern auch eine Einführung in die Sprache C++ zu geben. Dazu gehört die Vermittlung des Verständnisses, wie Templates, Strings, Container, smarte Pointer und anderes funktionieren und zusammenwirken. Wenn die Beispiele verstanden sind, ist es jedoch sinnvoll, sie nicht weiter zu entwickeln, sondern die vorgefertigten Komponenten der Standardbibliothek zu benutzen, weil Letztere mehr bieten und zudem durch die Compilerhersteller gepflegt werden.

Ein ebenso wichtiger Aspekt ist die Wartbarkeit von Programmen: Standardkonforme Programme können durch andere mit weniger Aufwand gepflegt werden, weil sich niemand in Spezialbibliotheken einarbeiten muss, wobei die Kenntnis des Standards natürlich vorausgesetzt wird. Aus diesem Grund sind in der Tabelle 26.3 die Beispielklassen dieses Buchs den in etwa vergleichbaren Komponenten der C++-Standardbibliothek gegenübergestellt.

Tabelle 26.3: Beispielklassen und die Standardbibliothek

Beispielklasse dieses Buchs	Abschnitt, Seite	etwa vergleichbare Standardkomponente	Standard-Header	siehe Seite
`MeinString`	6.1, 233	`string`	`<string>`	841
`Liste`	11.3, 404	`list`	`<list>`	772
`Warteschlange`	7.12, 297	`queue`	`<queue>`	777
`Stack`	6.3.2, 249	`stack`	`<stack>`	776
`SmartPointer`	9.5, 339	`shared_ptr`	`<memory>`	855
`Vektor`	9.2, 323	`vector`	`<vector>`	770

Die Vergleichbarkeit ist nicht nur beschränkt durch die umfangreichere und teilweise etwas andere Funktionalität der Standardbibliothek. Insbesondere wird in den Klassen der Standardbibliothek aus Geschwindigkeitsgründen häufig auf Prüfungen zur Laufzeit verzichtet. Zum Beispiel gibt es keine Exception bei der Klasse `shared_ptr`, wenn ein nicht-initialisierter Zeiger dereferenziert werden soll. Es gibt auch keine Laufzeitfehlermeldung, wenn ein außerhalb des zulässigen Bereichs liegender Index beim Zugriff auf ein `vector`-Objekt verwendet wird. Es steht dem Benutzer der Bibliothek jedoch frei, `vector::at(size_t)` aufzurufen oder Erweiterungen vorzunehmen. So kann eine Vektor-Klasse entworfen werden, deren Indexoperator eine Prüfung vornimmt, wie in Abschnitt 9.2 gezeigt.

Tipp

Eine gute Referenz zur C++-Standardbibliothek finden Sie unter *http://stdcxx.apache.org/doc/stdlibref/*.