O'REILLY®

C++

Standard-bibliothek

kurz & gut

O'REILLYS TASCHENBIBLIOTHEK

Rainer Grimm

C++-Standardbibliothek

kurz & gut

Rainer Grimm

O'REILLY®

Beijing · Cambridge · Farnham · Köln · Sebastopol · Tokyo

Die Informationen in diesem Buch wurden mit größter Sorgfalt erarbeitet. Dennoch können Fehler nicht vollständig ausgeschlossen werden. Verlag, Autoren und Übersetzer übernehmen keine juristische Verantwortung oder irgendeine Haftung für eventuell verbliebene fehlerhafte Angaben und deren Folgen. Alle Warennamen werden ohne Gewährleistung der freien Verwendbarkeit benutzt und sind möglicherweise eingetragene Warenzeichen. Der Verlag richtet sich im Wesentlichen nach den Schreibweisen der Hersteller. Das Werk einschließlich aller seiner Teile ist urheberrechtlich geschützt. Alle Rechte vorbehalten einschließlich der Vervielfältigung, Übersetzung, Mikroverfilmung sowie Einspeicherung und Verarbeitung in elektronischen Systemen.

Kommentare und Fragen können Sie gerne an uns richten:
O'Reilly Verlag
Balthasarstr. 81
50670 Köln
Tel.: 0221/9731600
Fax: 0221/9731608
E-Mail: kommentar@oreilly.de

Copyright:
© 2015 O'Reilly Verlag GmbH & Co. KG
1. Auflage 2015

Die Darstellung eines Meerschweinchens im Zusammenhang mit dem Thema C++ ist ein Warenzeichen von O'Reilly & Associates, Inc.

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.

Lektorat: Alexandra Follenius, Köln
Fachgutachten: Karsten Ahnert, Guntram Berti, Dmitry Ganyushin, Sven Johannsen, Torsten Robitzki und Bart Vandewoestyne
Korrektorat: Sibylle Feldmann, Düsseldorf
Umschlaggestaltung: Michael Oreal, Köln
Produktion: Karin Driesen, Köln
Satz: Reemers Publishing Services GmbH, Krefeld, www.reemers.de
Belichtung, Druck und buchbinderische Verarbeitung:
fgb freiburger graphische betriebe, ww.fgb.de

ISBN 978-3-95561-968-8

Dieses Buch ist auf 100 % chlorfrei gebleichtem Papier gedruckt.

Inhalt

KAPITEL 1
Einführung

C++-Standardbibliothek – kurz & gut ist eine Schnellreferenz zur Standardbibliothek des C++-Standards C++11. Der internationale Standard ISO/IEC 14882:2011 umfasst gut 1.300 Seiten und wurde 2011 veröffentlicht, also 13 Jahre nach dem bisher einzigen C++-Standard C++98. Formal betrachtet, ist zwar C++03 ein weiterer C++-Standard, der 2003 verabschiedet wurde. C++03 hat aber nur den Charakter einer technischen Korrektur.

Ziel dieser Kurzreferenz ist es, die Standardbibliothek von C++ kompakt vorzustellen. Im O'Reilly Verlag ist auch ein Buch zur C++-Kernsprache in der Reihe Taschenbibliothek erschienen. Ein Buch zur C++-Standardbibliothek setzt die Features der C++-Kernsprache voraus. Gegebenenfalls werde ich Features der Kernsprache vorstellen, um die Funktionalität der Standardbibliothek besser darstellen zu können. Beide Bücher dieser Reihe sind in Stil und Umfang sehr ähnlich und ergänzen sich ideal. 2014, beim Schreiben dieses Werks, wurde eine Ergänzung des C++11-Standards verabschiedet: C++14. In diesem Buch werde ich auf einige Neuerungen für C++ eingehen, die C++14 für die Standardbibliothek mit sich bringt.

Dieses Buch wurde für den Leser geschrieben, der eine gewisse Vertrautheit mit C++ besitzt. Dieser C++-Programmierer wird aus der konzentrierten Referenz der Standardbibliothek von C++ den größten Nutzen ziehen. Wenn C++ für Sie hingegen noch neu ist, sollten Sie im ersten Schritt ein Lehrbuch über C++ dieser Kurzreferenz vorziehen. Haben Sie das Lehrbuch aber gemeistert, hilft Ihnen dieses Werk mit den vielen kurzen Codebeispielen, die Kom-

ponenten der Standardbibliothek von C++ in einem weiteren Schritt sicher anzuwenden.

Konventionen

Mit diesen wenigen Konventionen sollte das Buch leichter lesbar sein.

Typografie

In dem Buch werden die folgenden typografischen Konventionen verwendet:

Kursiv
: Diese Schrift wird für Dateinamen und Hervorhebungen verwendet.

`Nichtproportionalschrift`
: Diese Schrift wird für Code, Befehle, Schlüsselwörter und Namen von Typen, Variablen, Funktionen und Klassen verwendet.

Quellcode

Obwohl ich kein Freund von `using`-Anweisungen und `using`-Deklarationen bin, da sie die Herkunft einer Bibliotheksfunktion verschleiern, werde ich gegebenenfalls in den Codebeispielen davon Gebrauch machen, und zwar einfach, weil die Länge einer Seitenzeile beschränkt ist. Es wird aber immer aus dem Codebeispiel hervorgehen, ob ich eine `using`-Deklaration (`using std::cout;`) oder eine `using`-Anweisung (`using namespace std;`) verwendet habe.

In den Codebeispielen werde ich nur die Header-Datei verwenden, dessen Funktionalität in dem Kapitel dargestellt wird.

Wahrheitswerte werde ich in den Ausgaben der Codebeispiele immer als `true` oder `false` darstellen, auch wenn das `std::boolalpha`-Flag (siehe *In diesem Buch werden nur Manipulatoren als Formatspecifier verwendet* (siehe Tipp Seite 178)) in dem Beispiel nicht verwendet wird.

Wert versus Objekt

Built-in-Datentypen, die C++ von C geerbt hat, nenne ich der Einfachheit halber Werte. Anspruchsvollere Datentypen, die oft *Built-in*-Datentypen enthalten, nenne ich Objekte. Dies sind in der Regel benutzerdefinierte Datentypen oder auch Container.

Index

Da die Namen der Standardbibliothek mit dem Namensraum `std` beginnen, verwende ich diesen im Index der Einfachheit halber nicht, sodass zum Beispiel die Information zum Vektor `std::vector` im Index unter `vector` zu finden ist.

Danksagungen

Ich möchte Alexandra Follenius, meiner Lektorin bei O'Reilly, für ihre Unterstützung und Anleitung bei der Arbeit mit diesem Buch danken. Danke vor allem aber auch an Karsten Ahnert, Guntram Berti, Dmitry Ganyushin, Sven Johannsen, Torsten Robitzki und Bart Vandewoestyne, die sich die Zeit genommen haben, das Manuskript auf sprachliche und insbesondere inhaltliche Fehler zu durchleuchten.

C++ versus C++11

Wer könnte C++11 besser charakterisieren als Bjarne Stroustrup, Erfinder von C++:

> Surprisingly, C++11 feels like a new language: The pieces just fit together better than they used to and I find a higher-level style of programming more natural than before and as efficient as ever.
>
> Bjarne Stroustrup, http://www.stroustrup.com/C++11FAQ.html

Bjarne Stroustrup hat recht. C++11 fühlt sich wie eine neue Sprache an, denn C++11 hat einiges gegenüber klassischem C++ zu bieten. Dies trifft nicht nur auf die Kernsprache, dies triff vor allem auch auf die verbesserte und deutlich erweiterte Standardbibliothek zu. Die Bibliothek mit den regulären Ausdrücken für die mächtigere Ver-

arbeitung von Texten, die Type-Traits-Bibliothek, um Typinformationen zu erhalten, zu vergleichen oder zu modifizieren, sowie die neue Zufallszahlen- oder Zeitbibliothek sind nun genauso auf der Habenseite von C++ wie die erweiterten Smart Pointer für die explizite Speicherverwaltung oder die neuen Container `std::array` und `std::tuple`. Neu ist vor allem aber auch, dass sich C++ zum ersten Mal mit der Multithreading-Bibliothek der Existenz von mehreren Threads bewusst ist.

KAPITEL 2

Die Standardbibliothek

Die C++-Standardbibliothek besteht aus vielen Komponenten. Dieses Kapitel gibt Ihnen auf der einen Seite einen kurzen Überblick darüber und zeigt zum anderen, wie die bestehende Funktionalität genutzt werden kann.

Die Chronologie

C++ und damit auch deren Standardbibliothek besitzen eine lange Geschichte. Sie beginnt in den 80er-Jahren des letzten Jahrtausends und endet vorerst 2014. Wer die Softwareentwicklung kennt, weiß, wie schnell die Branche tickt. Daher sind gut 30 Jahre ein sehr langer Zeitraum. So nimmt es nicht Wunder, dass deren erste Komponenten wie die I/O-Streams mit einem anderen Verständnis von guter Softwareentwicklung entworfen wurden als die moderne Standard Template Library (STL). Dieser Wandel in der Softwareentwicklung der letzten gut 30 Jahre, der sich in der C++-Standardbibliothek widerspiegelt, ist auch ein Wandel in der Art und Weise, wie Softwareprobleme gelöst werden. So startete C++ als objektorientierte Sprache, erweiterte sich insbesondere durch die STL der generischen Programmierung und nimmt mittlerweile immer mehr Impulse der funktionalen Programmierung auf.

Die erste C++-Standardbibliothek aus dem Jahr 1998 enthielt im Wesentlichen drei Komponenten. Das waren zum einen die bereits zitierten IO-Streams für den Umgang mit Dateien, das war zum anderen die Stringbibliothek, und das war die Standard Template

Library, die es auf generische Weise erlaubt, Algorithmen auf Containern anzuwenden.

Weiter ging es 2005 mit dem Technical Report 1 (TR1). Diese C++-Bibliothekserweiterung ISO/IEC TR 19768 war zwar kein offizieller Standard, enthielt aber viele Bibliothekskomponenten, die nahezu alle in den C++11-Standard aufgenommen wurden. So enthielt TR1 Bibliotheken zu regulären Ausdrücken, Smart Pointern, Hashtabellen sowie die Zufallszahlen- und Zeitbibliothek.

Neben der Standardisierung von TR1 enthält die Erweiterung eine vollständig neue Bibliothekskomponente: die Multithreading-Bibliothek.

Wie geht's weiter? Mit C++17 und C++22 sind zwei neue C++-Standards geplant. Diese werden vor allem neue Bibliotheken mit sich bringen. Zum jetzigen Zeitpunkt zeichnet es sich ab, dass C++ um die Unterstützung von Dateisystemen, Netzwerkprogrammierung und Modulen erweitert wird. Templates werden mit Concepts Lite ein Typsystem erhalten. Weitere High-Level-Bibliotheken zum Multithreading – insbesondere Transcational Memory – werden folgen.

Überblick

C++ enthält mittlerweile viele Bibliotheken. Da ist es nicht immer ganz einfach, die richtige Bibliothek für seinen Anwendungsfall auf Anhieb zu finden. Genau diese Suche soll dieser kurze Überblick erleichtern.

Praktische Werkzeuge

Praktische Werkzeuge, oder auch auf Englisch *Utilities*, sind Bibliotheken, die einen allgemeinen Fokus besitzen und daher in vielen Kontexten angewandt werden können.

Zu diesen praktischen Bibliotheken gehören Funktionen zum Berechnen des Minimums oder Maximums von Werten, aber auch Funktionen zum Tauschen oder Verschieben von Werten.

Weiter geht es mit dem praktischen Funktionspaar `std::function`/`std::bind`. Während `std::bind` es erlaubt, neue Funktionen aus bestehenden Funktionen zu erzeugen, ermöglicht es `std::function`, diese Funktionen an eine Variable zu binden und aufzurufen.

Mit `std::pair` und seiner Verallgemeinerung `std::tuple` lassen sich heterogene Paare und Tupel beliebiger Länge bilden.

Sehr praktisch sind die Referenz-Wrapper `std::ref` und `std::cref`, um einfach einen Referenz-Wrapper um eine Variable zu erzeugen, die im zweiten Fall konstant ist.

Das Highlight bei den praktischen Werkzeugen sind aber ohne Zweifel die Smart Pointer. Erlauben sie es doch, explizite Speicherverwaltung in C++ umzusetzen. Während `std::unique_ptr` den Lebenszyklus einer Variablen explizit verwaltet, bietet der Smart Pointer `std::shared_ptr` geteilte Besitzverhältnisse an. Dabei setzt er Referenz-Counting ein, um die Lebenszeit seiner Ressource zu managen. Zyklische Referenzen, das klassische Problem des Referenz-Countings, löst der dritte Smart Pointer im Bunde: `std::weak_ptr`.

Die Type-Traits-Bibliothek erlaubt es, Typeigenschaften abzufragen, Typen zur Übersetzungszeit zu vergleichen und sogar zu modifizieren.

Die Zeitbibliothek ist ein wichtiger Bestandteil der Multithreading-Fähigkeit von C++. Sie ist aber auch sehr praktisch für einfache Performancemessungen.

Die Standard Template Library

Die Standard Template Library (STL) besteht im Wesentlichen aus drei Komponenten. Das sind zum einen die Algorithmen und die Container, auf denen sie wirken, das sind zum anderen die Iteratoren, die das Bindeglied zwischen den Algorithmen und Containern darstellen. Die Abstraktion der generischen Programmierung er-

laubt es, Algorithmen und den Container auf einzigartige Weise zu kombinieren. Dabei stellen die Algorithmen und Container minimale Bedingungen an ihre verwendeten Typen.

Die C++-Standardbibliothek enthält eine reiche Kollektion an Containern. Auf der obersten Ebene sind es sequenzielle und assoziative Container. Die assoziativen Container lassen sich wiederum in geordnete und ungeordnete assoziative Container klassifizieren.

Jeder der sequenziellen Container besitzt sein besonderes Einsatzgebiet. In 95 % der Anwendungsfälle ist aber `std::vector` die richtige Wahl. So lässt sich seine Größe zur Laufzeit anpassen, er verwaltet seinen Speicher automatisch und bietet dies noch in Kombination mit herausragender Performance an. Im Gegensatz dazu unterstützt `std::array` als einziger sequenzielle Container keine Anpassung seiner Größe zur Laufzeit. Er ist sowohl auf Performance als auch auf minimale Speicheranforderungen getrimmt. Während `std::vector` für das Einfügen von Elementen an seinem Ende optimiert ist, kann dies `std::deque` auch am Anfang. Mit `std::list` als doppelt verkettete und `std::forward_list` als einfach verkettete Liste besitzt C++ zwei weitere Container, die Operationen auf beliebigen Stellen im Container mit hoher Performance ermöglichen.

Assoziative Container sind Container von Schlüssel/Wert-Paaren. Dabei bieten sie ihre Werte über ihre Schlüssel an. Typische Anwendungsfälle für assoziative Container sind Telefonbücher. Mit dem Schlüssel *Familienname* lässt sich einfach der Wert *Telefonnummer* ermitteln. C++ bietet acht verschiedene assoziative Container an: Da sind zum einen die assoziativen Container, deren Schlüssel geordnet sind: `std::set`, `std::map`, `std::multiset` und `std::multimap`. Und da sind zum anderen die ungeordneten assoziativen Container: `std::unordered_set`, `std::unordered_map`, `std::unordered_multiset` und `std::unordered_multimap`.

Zuerst zu den geordneten assoziativen Containern. Der Unterschied zwischen `std::set` und `std::map` ist, dass Ersterer keinen assoziierten Wert besitzt, der Unterschied zwischen `std::map` und `std::multimap` ist dass Letzterer mehrere gleiche Schlüssel erlaubt. Die Namenskonventionen gelten auch für die ungeordneten assoziati-

ven Container, die den geordneten sehr ähnlich sind. Der wesentliche Unterschied zwischen den geordneten und den ungeordneten assoziativen Containern ist aber deren Performance. Während die geordneten assoziativen Container eine Zugriffszeit zusichern, die logarithmisch von der Anzahl ihrer Elemente abhängt, erlauben die ungeordneten assoziativen Container im Durchschnittsfall eine konstante Zugriffszeit. Für `std::map` gilt das Gleiche wie für `std::vector`. Er ist in 95 % der Anwendungsfälle die erste Wahl für einen assoziativen Container.

Container-Adapter bieten ein vereinfachtes Interface für die sequenziellen Container an. C++ kennt den `std::stack`, `std::queue` und `std::priority_queue`. Damit lassen sich die entsprechenden Datenstrukturen implementieren.

Iteratoren sind das Bindeglied zwischen den Containern und den Algorithmen. Sie werden durch die Container direkt erzeugt. Als verallgemeinerte Zeiger erlauben sie je nach Containertyp das Vorwärts- sowie das Vor- und Rückwärts-Iterieren in dem Container oder sogar den wahlfreien Zugriff. Durch Iterator-Adapter können Iteratoren direkt mit einem Stream interagieren.

Neben dem reichen Satz an Containern enthält die STL über 100 Algorithmen, die auf Elemente oder Bereichen eines Containers wirken. Diese Bereiche werden immer durch zwei Iteratoren definiert. Der erste Iterator definiert dabei den Anfang, der zweite das Ende des Bereichs. Dabei gehört dieser sogenannte *Enditerator* selbst nicht mehr zum Bereich, sondern zeigt auf das Element direkt hinter dem Bereich.

Die Algorithmen spannen einen großen Anwendungsbereich auf. So lassen sich mit ihnen Elemente finden und zählen, Bereiche finden, vergleichen oder auch transformieren. Weiter geht es mit Algorithmen, die Elemente erzeugen, ersetzen oder auch entfernen. Auch für die Aufgaben, einen Container zu sortieren, zu partitionieren, sein Minimum und das Maximum zu bestimmen und seine Elemente zu permutieren, gibt es eine Vielzahl von verschiedenen Algorithmen. Viele Algorithmen können über aufrufbare Einheiten wie Funktionen, Funktionsobjekte oder auch Lambda-Funktionen parametrisiert werden. Damit ist es möglich, flexible Kriterien für die Suche

von Elementen oder das Transformieren von Elementen zu spezifizieren und damit die Mächtigkeit der Algorithmen deutlich zu steigern.

Numerik

Für die Numerik besitzt C++ zwei Bibliotheken. Das ist zum einen die Zufallszahlenbibliothek, und zum anderen sind es die mathematischen Funktionen, die C++ von C geerbt hat.

Die Zufallszahlenbibliothek besteht aus zwei Komponenten. Das sind zum einen die Zufallszahlenerzeuger und zum anderen die Zufallszahlenverteiler. Während die Zufallszahlenerzeuger einen Zahlenstrom zwischen einem Minimum- und einem Maximumwert erzeugt, bildet die Zufallszahlenverteilung diesen auf die Verteilung ab.

Dank C besitzt C++ viele mathematische Standardfunktionen. Dies sind logarithmische, expotenzielle oder auch trigonometrische Funktionen.

Textverarbeitung

Mit Strings und regulären Ausdrücken besitzt C++ zwei mächtige Bibliotheken, um Texte komfortabel zu verarbeiten.

Der `std::string` enthält einen reichen Satz an Methoden, um diesen zu analysieren oder zu modifizieren. Da er einem `std::vector` von Zeichen sehr ähnlich ist, können auch die Algorithmen der STL auf ihn angewandt werden. Als Nachfolger der C-Strings ist er deutlich sicherer, denn er verwaltet seinen Speicher automatisch.

Reguläre Ausdrücke sind eine Sprache zum Beschreiben von Zeichenmustern. Diese Zeichenmuster lassen sich mit der Reguläre-Ausdrücke-Bibliothek in C++ dazu verwenden, zu bestimmen, ob ein Zeichenmuster einem Text entspricht oder ob ein Zeichenmuster in einem Text ein- oder mehrmals vorkommt. Selbst Zeichenmuster lassen sich in einem Text ersetzen.

Ein- und Ausgabe

Mit den I/O-Streams steht in C++ schon lange eine Bibliothek zur Verfügung, die es erlaubt, mit der Außenwelt zu kommunizieren.

Kommunikation heißt in diesem konkreten Fall, dass der Extraktor (>>) erlaubt, von einem Eingabestream formatiert oder unformatiert zu lesen, und dass der Einfüger (<<) ermöglicht, auf einen Ausgabestream die Daten zu schieben. Diese können mit Manipulatoren formatiert werden.

Die Streamklassen bilden eine ausgefeilte Klassenhierarchie. Dabei sind zwei Streamklassen von besonderer Bedeutung. Zum einen ermöglichen Stringstreams, Strings und Stream miteinander interagieren zu lassen, zum anderen ermöglichen Dateistreams, Dateien einfach zu lesen und zu schreiben. Der Zustand eines Streams wird über Flags ausgedrückt, die gelesen und manipuliert werden können.

Durch das Überladen des Ein- und Ausgabeoperators interagiert ein eigener Datentyp wie ein Built-in-Datentyp mit der Außenwelt.

Multithreading

Mit dem 2011 verabschiedeten aktuellen C++-Standard C++11 erhielt C++ eine Multithreading-Bibliothek. Diese enthält die elementaren Bestandteile wie atomare Variablen, Threads, Locks und Bedingungsvariablen, auf denen zukünftige C++-Standards weitere Abstraktionen anbieten werden. Gegenüber diesen Bausteinen kennt C++11 aber auch Tasks, die einen deutlich abstrakteren Umgang mit mehreren Threads ermöglichen.

Auf Low-Level-Ebene besitzt C++11 zum ersten Mal ein Speichermodell und atomare Variablen. Beide zusammen sind die Grundlage dafür, dass Multithreading-Programme ein definiertes Verhalten ermöglichen.

Ein Thread startet in C++ sofort seine Arbeit. Er kann im Vorder- oder auch im Hintergrund laufen, seine Daten per Kopie oder per Referenz annehmen.

Der Zugriff auf von Threads gemeinsam genutzten Daten muss koordiniert werden. Diese Koordination bietet C++ mit Mutexen und Locks in verschiedenen Variationen an. Oft ist es aber ausreichend, die Daten sicher zu initialisieren, da sie während ihrer ganzen Lebenszeit als Konstante verwendet werden.

Thread-lokale Daten sind Daten, für die jeder Thread eine eigene Kopie besitzt. Damit kann es nicht zu Konflikten mit anderen Threads kommen.

Bedingungsvariablen sind eine klassische Lösung dafür, Sender-Empfänger-Arbeitsabläufe zu implementieren. Die Idee ist, dass der Sender signalisiert, dass er mit seiner Arbeit fertig ist, sodass der Empfänger mit seiner Arbeit beginnen kann.

Tasks sind Threads sehr ähnlich. Während bei einem Thread der Programmierer explizit diesen erzeugt, übernimmt das bei einem Task die C++-Laufzeit. Tasks lassen sich am einfachsten als geteilte Datenkanäle vorstellen. Ein *Promise* schiebt sein Datum in den Datenkanal, das ein *Future* abholen kann. Dieses Datum kann ein Wert, eine Ausnahme oder nur eine Benachrichtigung sein.

Bibliotheken verwenden

Um eine Bibliothek in einer ausführbaren Datei zu verwenden, sind in der Regel drei Schritte notwendig. Zuerst müssen die Header-Dateien durch `#include`-Anweisungen eingebunden werden. Damit sind die Namen der Bibliothek dem Compiler bekannt. Da sich die Header-Dateien der C++-Standardbibliothek in dem Namensraum `std` befinden, können ihre Namen im zweiten Schritt entweder voll qualifiziert verwendet oder müssen in den globalen Namensraum integriert werden. Zuletzt gilt es noch, dem Linker die zusätzlichen Bibliotheken anzugeben. Dieser Schritt kann in der Regel bei der C++-Standardbibliothek entfallen.

Header-Dateien einbinden

Durch die `#include`-Anweisung bindet der Präprozessor eine andere Datei, in der Regel eine Header-Datei, ein. Dabei werden die Header-Dateien in spitze Klammern eingeschlossen:

```
#include <iostream>
#include <vector>
```

WARNUNG

Da die Compiler-Implementierer frei sind, in ihren Header-Dateien zusätzliche Header-Dateien einzubinden, ist es möglich, dass Ihr Programm bereits alle notwendigen Header-Dateien implizit enthält, obwohl Sie diese nicht explizit angefordert haben. Sie sollten sich auf dieses Verhalten aber nicht verlassen, sondern immer alle Header-Dateien explizit anfordern, die die entsprechende Bibliothek benötigt. Denn bei einem Upgrade des Compilers oder einer Portierung des Programms auf eine andere Plattform ist die Gefahr groß, dass sich Ihr Programm infolge fehlender Header-Dateien nicht mehr übersetzen lässt.

Namensräume verwenden

Die Namen eines Namensraums können qualifiziert, unqualifiziert oder mit einem Alias verwendet werden.

Namen qualifiziert verwenden

Bei der qualifizierten Verwendung eines Namens werden diese genau so verwendet, wie sie definiert wurden. Jedem Namensraum wird dabei der Bereichsauflöser :: vorangestellt. Mehrere Bibliotheken der C++-Standardbibliothek verwenden mehrere eingebettete Namensräume. Diese verschachtelten Namensräume müssen bei der qualifizierten Anwendung spezifiziert werden:

```
#include <iostream>
#include <chrono>
...
std::cout << "Hello world:" << std::endl;
auto timeNow= std::chrono::system_clock::now();
```

Namen unqualifiziert verwenden

Namen können in C++ mit der using-Deklaration und der using-Anweisung verwendet werden.

using-Deklaration

Eine using-Deklaration fügt einen Namen zu dem Sichtbarkeitsbereich hinzu, in dem sich die using-Deklaration selbst befindet:

```
#include <iostream>
#include <chrono>
...
using std::cout;
using std::endl;
using std::chrono::system_clock;
...
cout << "Hello world:" << endl;  // unqualifizierter Name
auto timeNow= now();             // unqualifizierter Name
```

Die Verwendung der using-Deklaration besitzt die folgenden Konsequenzen:

- Ein Compiler-Fehler tritt auf, wenn der gleiche Name auch andernorts im gleichen Sichtbarkeitsbereich deklariert wird.
- Wenn der gleiche Name in einem umgebenden Sichtbarkeitsbereich deklariert wird, wird dieser durch die using-Deklaration verborgen.

using-Anweisung

Die using-Anweisung erlaubt es, alle Namen aus einem Namensraum ohne Qualifikation zu verwenden:

```
#include <iostream>
#include <chrono>
...
using namespace std;
...
cout << "Hello world:" << endl;  // unqualifizierter Name
auto timeNow= chrono::system_clock::now(); //teilqualifizierter
                                           //Name
```

Eine `using`-Anweisung fügt keinen Namen zum aktuellen Sichtbarkeitsbereich hinzu, sondern macht diese nur von dort aus erreichbar. Das bedeutet insbesondere:

- Wenn ein Name andernorts im lokalen Sichtbarkeitsbereich deklariert wird, wird der Name im Namensraum von der lokalen Deklaration verborgen.
- Ein Name im Namensraum verbirgt einen im umgebenden Sichtbarkeitsbereich deklarierten gleichen Namen.
- Ein Compiler-Fehler tritt auf, wenn der gleiche Name aus mehreren Namensräumen sichtbar gemacht wird oder wenn ein Name im Namensraum einen Namen im globalen Namensraum verbirgt.

WARNUNG

`using`-Anweisungen sollten mit Vorsicht in Quelldateien verwendet werden, denn durch ein `using namespace std` werden alle Namen aus `std` sichtbar. Damit werden gegebenenfalls auch Namen sichtbar, die Namen im lokalen oder umgebenden Namensraum *zufällig* verbergen.

`using`-Anweisungen sollten nicht in Header-Dateien verwendet werden. Eine `using namespace std` führt dazu, dass durch das Einbinden der Header-Datei automatisch alle Namen aus `std` sichtbar sind.

Namensraum-Alias

Ein Namensraum-Alias deklariert einen alternativen Namen für einen Namensraum. Ein neuer Name für einen bestehenden Namensraum bietet sich bei einem sehr langen Namen oder bei eingebetteten Namensräumen an:

```
#include <chrono>
...
namespace sysClock= std::chrono::system_clock;
```